Lightweight Specifications for Parallel Correctness

Jacob Burnim

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-226
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-226.html

December 5, 2012

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
05 DEC 2012		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Lightweight Specifications for Parallel Correctness	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

With the spread of multicore processors, it is increasingly necessary for programmers to write parallel software. Yet writing correct parallel software with explicit multithreading remains a di cult undertaking. Though many tools exist to help test, debug, and verify parallel programs, such tools are often hindered by a lack of any speci cation from the programmer of the intended, correct parallel behavior of his or her software. In this dissertation, we propose three novel lightweight speci cations for the parallelism correctness of multithreaded software: semantic determinism, semantic atomicity, and nondeterministic sequential speci cations for parallelism correctness. Our determinism speci - cations enable a programmer to specify that any run of a parallel program on the same input should deterministically produce the same output, despite the nondeterministic interleaving of the program's parallel threads of execution. Key to our determinism speci cations are our proposed bridge predicates | predicates that compare pairs of program states from di erent executions for semantic equivalence. Our atomicity speci cations use bridge predicates to generalize traditional atomicity, enabling a programmer to specify that regions of a parallel or concurrent program are, at a high-level, free from harmful interference by other threads. Finally, our nondeterministic sequential (NDSeq) speci cations enable a programmer to completely specify the parallelism correctness of a multithreaded program with a sequential but nondeterministic version of the program and, further, enable a programmer to test, debug and verify functional correctness sequentially, on the nondeterministic sequential program. We show that our lightweight speci cations for parallelism correctness enable us to much more e ectively specify, test, debug, and verify the use of parallelism in multithreaded software independent of complex and fundamentally-sequential functional correctness. We show that we can easily write determinism, atomicity, and nondeterministic sequential (NDSeq) speci cations for a number of parallel Java benchmarks. We propose novel testing techniques for checking that a program conforms to its determinism, atomicity, or nondeterministic sequential speci cation, and we apply these techniques to nd a number of parallelism errors in our benchmarks. Further, we propose techniques for automatically inferring a likely determinism or NDSeq speci cation for a parallel program, given a handful of representative executions.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**175**	

Lightweight Specifications for Parallel Correctness

by

Jacob Samuels Burnim

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Professor George Necula
Professor David Wessel

Fall 2012

Abstract

Lightweight Specifications for Parallel Correctness

by

Jacob Samuels Burnim

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

With the spread of multicore processors, it is increasingly necessary for programmers to write parallel software. Yet writing correct parallel software with explicit multithreading remains a difficult undertaking. Though many tools exist to help test, debug, and verify parallel programs, such tools are often hindered by a lack of any specification from the programmer of the intended, correct parallel behavior of his or her software.

In this dissertation, we propose three novel lightweight specifications for the parallelism correctness of multithreaded software: semantic determinism, semantic atomicity, and nondeterministic sequential specifications for parallelism correctness. Our determinism specifications enable a programmer to specify that any run of a parallel program on the same input should deterministically produce the same output, despite the nondeterministic interleaving of the program's parallel threads of execution. Key to our determinism specifications are our proposed bridge predicates — predicates that compare pairs of program states from different executions for semantic equivalence. Our atomicity specifications use bridge predicates to generalize traditional atomicity, enabling a programmer to specify that regions of a parallel or concurrent program are, at a high-level, free from harmful interference by other threads. Finally, our nondeterministic sequential (NDSeq) specifications enable a programmer to completely specify the parallelism correctness of a multithreaded program with a sequential but nondeterministic version of the program and, further, enable a programmer to test, debug, and verify functional correctness sequentially, on the nondeterministic sequential program.

We show that our lightweight specifications for parallelism correctness enable us to much more effectively specify, test, debug, and verify the use of parallelism in multithreaded software, independent of complex and fundamentally-sequential functional correctness. We show that we can easily write determinism, atomicity, and nondeterministic sequential (NDSeq) specifications for a number of parallel Java benchmarks. We propose novel testing techniques for checking that a program conforms to its determinism, atomicity, or nondeterministic sequential specification, and we apply these techniques to find a number of parallelism errors in our benchmarks. Further, we propose techniques for automatically inferring a likely determinism or NDSeq specification for a parallel program, given a handful of representative executions.

To my parents.

Contents

Acknowledgments

This dissertation would not have been possible without the support of many collaborators, colleagues, friends, and family. I am deeply grateful to all of you.

I would like to thank my advisor, Koushik Sen. I am grateful for Koushik's patient mentoring and teaching, for his savvy simplifying suggestions, for his constant enthusiasm and endless flow of ideas, and for his keen research sense.

I would also like to thank George Necula for chairing my qualifying exam committee and serving on my thesis committee. I am grateful for George's wise research perspective and for his excellent and compassionate guidance.

I thank Ras Bodik for his insightful questions and sage advice at my qualifying exam, and at many other talks and seminars. Thanks also to David Wessel for serving on my qualifying exam and dissertation committees.

I would like to thank my collaborators Tayfun Elmas, Nick Jalbert, Sudeep Juvekar, Madan Musuvathi, Shaz Qadeer, and Christos Stergiou. I am grateful for the chance to have worked with each of you. Thanks also to Tom Ball, Madan Musuvathi, and Shaz Qadeer for hosting me at Microsoft Research.

I am grateful to my many wonderful colleagues at Berkeley, including Shaon Barman, Derrick Coetzee, Joel Galenson, Benjamin Hindman, Thibaud Hottelier, Pallavi Joshi, Benjamin Lipshitz, Leo Meyerovich, Mayur Naik, Chang-Seo Park, and Philip Reames — many thanks for the research discussions, for the feedback on papers and talks, and for much more. Special thanks to Roxana Infante and Tammy Johnson for their tireless and heroic efforts on all of our behalf.

Many thanks to my roommates and dear friends Callum Lamb, Benjamin Lipshitz, and Meru Sadhu.

I would like to thank my mother, my father, and my brother. I am deeply grateful for all of their love and support.

And thanks to Rachel, for everything.

This work supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by Sun Microsystems (now Oracle) and by matching funding from UC MICRO (Award #08-113), by NSF Grants CNS-0720906, CCF-0747390, CCF-101781, CCF-1018729, and CCF-1018730, and by an National Defense Science & Engineering Graduate (NDSEG) Fellowship. Additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

Chapter 1

Introduction

The semiconductor industry has hit the power wall — performance of general-purpose single-core microprocessors can no longer be increased due to power constraints. Therefore, to continue to increase performance, the microprocessor industry is instead increasing the number of processing cores per die. The new "Moore's Law" is that the number of cores will double every generation, with individual cores going no faster [10].

The spread of multicore processors and the end of rapidly growing single-core performance is increasing the need for programmers to write parallel software. Yet writing correct parallel software with explicit multithreading remains a difficult undertaking. A programmer must ensure not only that each part of his or her program computes the correct result in isolation, but also that the uncontrolled and nondeterministic interleaving of the program's parallel threads cannot cause harmful interference, leading to incorrect final results. Thus, parallel programs suffer from a host of new kinds of errors, such as data races, atomicity violations, and deadlocks. The need to simultaneously reason about sequential functional correctness and the correctness of parallel interleavings poses a great challenge both for programmers writing, understanding, and testing their software and for tools analyzing and verifying such software.

Thus, there has been great interest in techniques and tools to better test parallel software and to automatically find parallelism errors. There has been much work on exposing errors in parallel programs by testing such programs under a wider variety of thread schedules, via stress testing, random scheduling [45, 140, 51, 131, 25], and systematic schedule exploration [72, 24, 155, 35, 105, 152, 26, 133] (i.e. explicit model checking). Further, a number of techniques have been developed to find parallel *bug patterns*, such as static [139, 59, 23, 54, 71, 49, 120, 107] and dynamic [44, 2, 130, 122, 38, 3, 132, 58, 136] techniques to detect and predict *data races*. Similarly, techniques have been developed to detect high-level races [122, 9] and non-atomic methods [64, 158, 76, 3, 57, 159, 12, 62, 52, 37, 157]. Recent work has proposed to combine these two approaches, first using a static or dynamic analysis to predict the presence of certain parallel bug patterns in a program, and then controlling or biasing the program's thread scheduling in order to create real executions in which the bug patterns occur. This approach, *active testing* [132, 85], has been successful in finding and reproducing,

e.g., races [132], atomicity violations [116, 117, 138, 91], and concurrency-related memory errors [167, 32] and assertion violations [168].

While such testing and program analysis tools have been very successful, they are often hindered by the following problem: In practice, it can be quite difficult to determine whether or not such a tool has truly found a program error. That is, when software testing leads to a program crash or an assertion failure, we can be certain that a real error has been found. Such tools, however, often find program behavior that is *suspicious*, but not unambiguously erroneous. For complex programs, such tools may observe hundreds of real data races or violations of method atomicity in program executions, none of which lead to assertion or test failures, uncaught exceptions, or other clear program errors. Such low-level, suspicious behaviors could be real errors, but they are often benign, having no effect on the high-level correctness of the program. Thus, parallelism correctness tools must either ignore these potential errors or must report them all to a user, forcing the user to laboriously determine whether or not each observed program behavior is correct.

We argue that, in order to effectively test and verify parallel software, we must have some kind of specification from the programmer of the intended, correct parallel behavior of his or her software. Unfortunately, traditional functional correctness specifications — e.g., formalizing the correct output of a program as a function of its input — can be very complex and challenging to write. (Imagine, for example, specifying the correct output of an application to render a fractal or of an application to compute a likely phylogenetic tree given genetic sequence data.) We believe there is a need for easier-to-write specifications that focus on just the correctness of a program's use of parallelism. Such specifications should allow us to specify, test, debug, and verify the parallelism correctness of a program without having to grapple with a program's complex and fundamentally-sequential functional correctness.

Contributions and Outline

In this dissertation, we propose three novel, lightweight specifications for the parallelism correctness of multithreaded programs: semantic determinism [29, 30, 31], semantic atomicity [28], and nondeterministic sequential (NDSeq) specifications for parallel correctness [27, 34, 33]. We claim that our lightweight specifications for parallelism correctness enable us to effectively specify, test, debug, and verify the use of parallelism in multithreaded software, independent of complex, sequential functional correctness.

Semantic Determinism Specifications

In Chapter 3, we describe our proposed semantic determinism assertions [29, 30], for specifying that two runs of a parallel program on the same input should give results that are *semantically* the same, despite any nondeterministic interleaving of the program's parallel threads of execution. Key to our determinism specifications are our proposed *bridge predicates* [29, 30] — predicates relating pairs of program states from different executions.

Bridge predicates allow us to simply specify, at a high level, the way in which the results of two different parallel executions should be equivalent. We show that we can easily specify this high-level parallelism correctness property for a variety of parallel Java benchmarks, including many for which we are unable to write any kind of useful functional correctness specification. Further, by combining our determinism assertion library with existing techniques [132, 85], we can effectively test our determinism specifications — e.g, identifying races that lead to erroneous nondeterministic behavior and separating these races from the many benign races found by previous techniques.

Automatically Inferring Likely Determinism Specifications

In Chapter 4, we propose DETERMIN], our algorithm [31] to dynamically infer a likely determinism specification for a parallel program, given a set of representative program executions. We show that the simpler form of determinism specifications, compared to that of traditional functional correctness specifications, enables us to efficiently compute useful determinism specifications consistent with a set of program executions. We implement our specification inference algorithm for parallel Java programs and apply our algorithm to the Java benchmarks used in Chapter 3. Given only a handful of representative executions, DETERMIN can automatically infer equivalent or better determinism specifications than our hand-written specifications for nearly all of our benchmarks. We argue that the inference of determinism specifications can aid in understanding and documenting the deterministic behavior of parallel programs. Moreover, an unexpected determinism specification can indicate to a programmer the presence of erroneous or unintended parallel behavior.

Semantic Atomicity Specifications

In Chapter 5, we describe our proposed semantic atomicity assertions [28], a generalization of traditional atomicity, which uses our bridge predicates to specify critical, high-level noninterference properties of parallel code. We also describe our novel technique [28] for testing semantic atomicity on *interruption-bounded* executions, and show that we can effectively specify and test semantic atomicity on a number of parallel and concurrent benchmarks, finding several known and unknown atomicity errors with no false positives, while previous, strict atomicity techniques would report many false warnings.

Nondeterministic Sequential Specifications

In Chapter 6, we describe our proposal for using a *nondeterministic* but *sequential* version of a program as a *complete* specification for the correctness of the program's use of parallelism. If we can verify or otherwise gain confidence in the parallelism correctness of such a program, we can then test, debug, and verify the program's functional correctness on a version of the program with controlled nondeterminism but with no interleaving of parallel threads. We propose a novel technique [34] for runtime checking of our nondeterministic sequential

(NDSeq) [27] specifications, and we show that we can use NDSeq specifications both to specify and to test the parallelism correctness of a number of parallel applications, including those with quite complex sequential behavior.

Further, we discuss future work to use our NDSeq specifications and our runtime checking technique to classify an erroneous program trace as containing either a sequential or a parallel error, and, in the former case, allowing the error to be debugged on a nondeterministic sequential trace, instead. And we discuss future work to statically verify parallelism correctness, independent of a program's complex and sequential functional correctness, by verifying — via reduction [93] — that every parallel program behavior is also a behavior of the nondeterministic sequential specification.

Automatically Inferring Likely NDSeq Specifications

In Chapter 7, we propose an algorithm [33], for automatically inferring a likely nondeterministic sequential (NDSeq) specification for a parallel program, given a set of representative program executions. Our technique combines dynamic data flow analysis and Minimum-Cost Boolean Satisfiability (MinCostSAT) solving to find a likely NDSeq specification for the correctness of a program's parallelism — an NDSeq specification that captures all nondeterminism of the observed parallel executions. We implement our technique for Java in a prototype tool called nDetermin. For a number of benchmarks, nDetermin infers equivalent or stronger NDSeq specifications than those we previously wrote manually.

Chapter 2

Overview of Lightweight Specifications for Parallelism Correctness

In this chapter, we give a high-level overview of our lightweight specifications for parallelism correctness, using a simple parallel program as running example. We present this motivating example in Section 2.1, and we discuss in Section 2.2 the challenges in checking the example's parallelism correctness. In Section 2.3, we will motivate and describe *semantic determinism specifications* on this running example. In Section 2.4, we will show how the inherent *nondeterministic* behavior of our running example necessitates a second kind of partial, high-level specification: *semantic atomicity*. Finally, in Section 2.5, we will show how to give a *complete* specification of parallelism correctness for our running example using a *nondeterministic sequential (NDSeq) specification*. We will also briefly discuss our approaches to testing that a program conforms to each of the above specifications, and we will motivate and describe our techniques for automatically inferring likely determinism specifications and nondeterministic sequential (NDSeq) specifications.

2.1 Running Example

Consider the generic, parallel branch-and-bound procedure given in Figure 2.1 on the next page. The search procedure finds a minimum-cost solution in a given solution space. Initially, the FIFO work-queue `queue` holds a single element representing the entire space to be searched. The procedure repeatedly retrieves a unit of work — i.e. a region of the solution space — from the `queue` and either: (1) prunes the region if the computed lower bound on its minimum-cost solution (`lower_bound_cost(work)`) exceeds the best cost found so far (Lines 2–3), (2) exhaustively searches for the minimal solution in the region when the region is smaller than some threshold (Lines 5–11), or (3) splits the regions into smaller pieces for later processing (Line 13). The output is a minimum-cost solution `min_cost_soln`, along with the cost `lowest_cost` of that solution.

```
 1:  coforeach (work in queue) {
 2:      if (lower_bound_cost(work) >= lowest_cost)
 3:          continue;
 4:      if (size(work) < THRESHOLD) {
 5:          soln = find_min_cost_solution(work);
 6:          synchronized (lock) {
 7:              if (cost(soln) < lowest_cost) {
 8:                  lowest_cost = cost(soln);
 9:                  min_cost_soln = soln;
10:              }
11:          }
12:      } else {
13:          queue.add(split(work))
14:      }
15:  }
```

Figure 2.1: Our running example — a generic, parallel branch-and-bound procedure for finding the minimum-cost solution in a given solution space. Variables `queue`, `lowest_cost`, and `min_cost_soln` are shared between parallel loop iterations, while variables `work` and `soln` are thread local. Not shown is the code to read the input, to initialize `queue`, and, at the end, to output `lowest_cost` and `min_cost_soln`.

Our example procedure is a *parallel* search: the `coforeach` construct spawns a new parallel iteration whenever `queue` contains a work item, `continue` ends its parallel iteration, and the loop terminates when all spawned iterations have finished and `queue` is empty. Further, the code synchronizes on lock `lock` to safely update shared variables `lowest_cost` and `min_cost_soln` in Lines 7–10. (The `queue` data structure must be thread-safe, as well.) Note, however, that procedures `lower_bound_cost` and `find_min_cost_solution` are *thread-local*, operating only on data private to the calling thread.

2.2 The Challenge of Parallelism Correctness

The use of parallelism in our example in Figure 2.1 is quite simple: a single parallel construct (`coforeach`) is used to run loop iterations in parallel, a thread-safe shared work `queue` is used, and one shared `lock` is used to protect writes to the two shared variables (`lowest_cost` and `min_cost_soln`). Thus, we claim that we should be able to test, debug, reason about, and verify the correctness of our example's use of parallelism in similarly simple way — in particular, without getting entangled in the complex details of the fundamentally-sequential functional correctness of the benchmarks.

As discussed in Chapter 1, there exists a wide variety of techniques and tools to automatically test and verify parallel programs. But we claim that such tools are not sufficient for checking the parallelism correctness for our running example. Such tools typically fall into two categories:

1. Many techniques aim to find parallelism errors in a completely automated way, with no additional specification from the user — for example, tools to detect *data races* or potential *atomicity violations*. (Note that we can think of such tools as using an implicit parallelism correctness specification — i.e., "a program with correct parallelism contains no data races".) But the issues reported by such tools are often not true parallelism errors.

   Such a tool might report, for example, that our branch-and-bound program has a *data race* — `lowest_cost` can be concurrently read at Line 2 and written at Line 8. Similarly, such a tool might assume that body of the `coforeach`-loop is intended to execute atomically, and then report that our example has a potential *atomicity violation* — as a parallel loop iteration may read `lowest_cost` at both Line 2 and Line 7, while another thread can modify `lowest_cost` in between these two reads. But, as we will show later in this chapter, neither the data race nor the apparent atomicity violation are bugs — the use of parallelism in our example program is correct.

2. To avoid the above problem of finding and reporting program behaviors that are suspicious but ultimately not erroneous, many techniques rely on some kind of functional correctness specification from the user. Such techniques then report an error only when a (potential) program execution has been found that violates the given specification. The specification can range from low-level partial correctness assertions such as the absence of null-pointer dereferences and other runtime errors, to unit tests (which, in essence, specify that, for a specific input, the program's output and behavior must satisfy certain assertions), to full functional correctness specifications that specify exactly what outputs are correct as a function of the program input.

   But it can be extremely difficult to write the kind of functional correctness specification that that is needed to effectively test or verify parallelism correctness with this approach. Imagine, for example, writing postconditions to specify the correct output of an application to render a fractal or of an application to compute a likely phylogenetic tree given genetic sequence data.

   For our branch-and-bound procedure, for example, writing a specification that the returned `min_cost_soln` is a valid solution, let alone a solution achieving the minimum cost, could be quite complex. Constructing even a single correct test input and output may also be quite challenging, especially given that we often need fairly large test inputs to exercise the most interesting and error-prone aspects of a parallel program. In particular, one of our benchmark programs (`phylogeny`), which we examine in later chapters, is a branch-and-bound search. We were unable to write any kind of useful traditional functional correctness specification or unit test for this benchmark.

   Note, however, the the above difficulties are all due to the *sequential* complexity of our example program. That is, while functions `find_min_cost_solution` and `lower_bound_cost` may be very complex — e.g., finding a solution to an integer programming problem and bounding such a solution with a linear programming relaxation

— these functions perform sequential computations on thread-local data. And any errors relating to these functions are *sequential* errors — i.e., a sequential version of our example program, in which the main loop was a simple, sequential `for`-loop, would have the same errors.

We claim there is a great need for easier-to-write specifications that focus on just the correctness of a program's use of parallelism. Thus, we propose two high-level and partial parallelism correctness specifications: *semantic determinism* and *semantic atomicity*. Further, we propose *nondeterministic sequential (NDSeq) specifications*, which allow us to give a *complete* specification of the parallelism correctness of a program, using a sequential but nondeterministic version of the program. In the remainder of this chapter, using our running example, we discuss how we can both effectively write, test, and automatically infer such specifications without having to grapple with the complex and fundamentally-sequential functional correctness.

2.3 Semantic Determinism Specification

A key correctness property for parallel programs is *determinism*. We say that a parallel computation is deterministic if it always produces the same answer (i.e. the same output or final result) when given the same input, no matter how the parallel threads of execution are nondeterministically interleaved. If we can show that a parallel program is deterministic, then we often have a strong signal that the program's use of parallelism is correct. That is, how can there be any parallelism errors if every different thread schedule produces the same result?

Thus, in Chapter 3 we propose *semantic determinism* specifications to allow a programmer to specify that the final result of a computation should be the same, despite the uncontrolled nondeterministic interleaving of the parallel threads. Formally, for a block P of parallel code, a programmer can write:

```
deterministic assume(Pre(s0, s0')) {
    P
} assert(Post(s, s'));
```

This specification asserts the following: Suppose P is executed twice with potentially different schedules, once from initial state s_0 and once from initial state s'_0, yielding final states s_1 and s'_1, respectively. Then, if the user-specified *precondition Pre* holds over s_0 and s'_0, then s and s' must satisfy the user-specified *postcondition Post*.

Note that this specification is quite different from a traditional functional correctness specification, in which the precondition and postcondition would each refer to a *single* program state — one at the beginning of P and one at the end of P. Such a traditional specification would have to specify correct behavior by relating the final state to the initial state. Our determinism specifications instead require only specifying *equivalence* between pairs of initial states and between pairs of final states.

For our example program in Figure 2.1, we can write the determinism specification:

```
deterministic assume(equalElements(queue, queue')) {
    ... code from Figure 2.1 ...
} assert(lowest_cost == lowest_cost');
```

That is, given two runs of our example code on `queue`s which contain equal regions of the solution space, we should get the same final output. Note the use of primed variables `queue'` and `lowest_cost'` in our determinism specification. These variables represents the values of variables `queue` and `lowest_cost` in the corresponding state from a different execution. As our precondition and postcondition each relate two different program states from different program executions, we call such a predicate a *bridge predicate*.

Our semantic determinism specifications have several advantages. Our determinism specification for our example program is quite simple to write, while, as we discussed in Section 2.2, writing a traditional functional correctness specification could be very difficult. The specification clearly captures a natural and important correctness property of our example program — if, for the same input, two different thread schedules could return answers with different costs, there would clearly be an error in the program.

Further, the use of bridge predicates in our determinism specifications allow a programmer to specify that different thread schedules need not produce bit-by-bit identical final states, but only need to produce results that are equivalent at the semantic level of his or her application. We argue in Chapter 3 that this makes our determinism specifications much more widely-applicable than previous, more strict approaches to specifying and checking determinism. We allow programmers to write these bridge predicates using standard Java, in order to take advantage of existing methods (i.e., `equals` methods) or to simply write custom methods for checking for semantic equivalence. In our example determinism specification, we use method `equalElements` to check `queue` and `queue'` for semantic equivalence — this method returns `true` if `queue` and `queue'` contain the same *set* of elements, irrespective of the order of the elements or of the internal structure or layout in memory of `queue` and `queue'`. (And the elements themselves are similarly compared via `equals` methods, rather than a low-level, bit-by-bit comparison.)

Note that, for our example program, we cannot use the specification:

```
deterministic assume(equalElements(queue, queue')) {
    ... code from Figure 2.1 ...
} assert(lowest_cost==lowest_cost'
           && min_cost_soln.equals(min_cost_soln');
```

Different thread schedules cause the solution space to be searched in different orders, changing which regions are pruned and changing which solution is found among multiple minimum-cost solutions. Thus, different executions can correctly return completely different final `min_cost_soln`. This nondeterminism is an intended and expected part of our example program. (It is possible to restructure the example to return a fully-deterministic final answer, but at a performance cost.)

Testing Semantic Determinism Specifications

Suppose that we want to test that our example program conforms to its determinism specification from the previous section:

```
deterministic assume(equalElements(queue, queue')) {
    ... code from Figure 2.1 ...
} assert(lowest_cost == lowest_cost');
```

Suppose further that we have several test inputs for the program. For each test input, suppose that we run our example program several times. Note that our determinism specification contains one specified *deterministic block*, around the main `coforeach`-loop in the example program. When each test execution reaches the end of the specified deterministic block, we would like to compare the final value of `lowest_cost` to the final values from other executions which started with equivalent values of `queue`.

In Chapter 3, we propose a simple technique for performing such testing of determinism specifications. We implement a determinism specification library for Java that, at the beginning and at the end of the execution of any deterministic block, serializes and records all program state that is referenced by the determinism precondition and postcondition. That is, in each of our runs of our example program, just before starting the `coforeach`-loop, our determinism specification library serializes and saves to disk the contents of `queue`. And, at the end of the `coforeach`-loop, our determinism library saves to disk the value of `lowest_cost`. Further, our library compares the saved values (`queue`, `lowest_cost`) to the values (`queue'`, `lowest_cost'`) saved in all previous test executions. If we find any pairs for which:

$$\texttt{queue.equals(queue')} \ \wedge \ (\texttt{lowest\_cost != lowest\_cost'})$$

then we report a *determinism violation*. That is, we warn the programmer, because we have found that the program is not semantically deterministic as intended — we have two different parallel executions that incorrectly produce nonequivalent results starting from equivalent initial states.

To make our determinism testing more effective, we can combine the checking of our determinism specification library with any existing technique for perturbing the thread scheduling of a parallel program in order to more quickly find and expose bugs. In Chapter 3, we combine our determinism testing with active testing [132, 85], which finds thread schedules with real data races, non-atomic methods, etc.

Our experiments with determinism specifications and active testing highlight a key advantage of using lightweight specifications for parallelism correctness. On our benchmarks, existing active testing techniques report 40 real data races, such as the data race between Line 2 and Line 8 of our example program (on variable `lowest_cost`). But only one of these data races leads to a violation of our determinism specifications. By writing determinism specifications, a programmer can quickly separate the real bugs (i.e., data races leading to incorrect, nondeterministic program results) from benign data races, such as the one in our example program, which are not actually parallelism errors.

Automatically Inferring Determinism Specifications

We show in Chapter 3 that our determinism specifications allow a programmer to effectively test and document the intended, high-level deterministic behavior of his or her application. But, to see any benefit, the programmer must first write a determinism specification for the target application. We could save programmer effort if we had a way to automatically write determinism specifications for parallel applications. Further, being able to automatically generate determinism specifications would enable automatic determinism checking, in which we first generated and then tested a semantic determinism specification for an application.

We show in Chapter 4 that, if we observe a handful of representative executions of a parallel program, it is possible to automatically infer a likely determinism specification for the program. The key insight is that, although the space of possible determinism specifications is infinite, in practice determinism specifications have a very simple structure: the precondition and the postcondition are conjunctions of a number of *equality predicates* — predicates comparing a single variable to its primed counterpart for equality or approximate equality. For example, for our branch-and-bound search, we would only consider determinism preconditions and postconditions containing predicates such as:

- `queue.equals(queue'`)
- `equalElements(queue, queue'`)
- `lowest_cost` $==$ `lowest_cost'`
- $|$`lowest_cost` $-$ `lowest_cost'`$| < \epsilon$
- `min_cost_soln.equals(min_cost_soln'`)

From representative executions, we can observe different `min_cost_soln` at the end of the `coforeach`-loop for executions with identical states at the start of the `coforeach`-loop. Thus, we see that the strongest postcondition consistent with our observations is:

`lowest_cost != lowest_cost'`

(If `lowest_cost` was a floating-point value, approximate equality might be needed, instead.)

Further, we observe that the weakest precondition that guarantees that this postcondition holds for all of our observed executions is:

`equalElements(queue, queue'`)

Thus, given a sufficiently representative set of program executions, we can automatically infer our determinism specification for our running example.

In Chapter 4, we describe DETERMIN, our proposed specification inference technique, and we show that, for nearly all of our benchmarks from Chapter 3, DETERMIN can produce determinism specifications equivalent to or better than those we wrote by hand. Further, one of the inferred specifications highlights an error in our previous, manual determinism specification for one of our benchmarks.

2.4 Semantic Atomicity Specifications

We find in Chapter 3 that our determinism specifications capture critical parallelism correctness properties for many applications. But, as discussed in the previous section, parallel applications can intentionally have nondeterministic behavior — e.g., our branch-and-bound example in Figure 2.1, which performs a nondeterministic search through its solution space. Thus, although we can use a determinism specification to specify that any parallel behavior that leads to non-equivalent final `lowest_cost` is an error, we cannot specify anything about parallelism errors that lead to incorrect `min_cost_soln`. For applications with such *algorithmic nondeterminism*, we need some additional other kind of lightweight specification in order to distinguish the nondeterministic interactions between parallel threads that are expected from erroneous parallel interference between threads.

Similarly, for concurrent data structures and other "open" parallel libraries, there is no primary parallel computation, with inputs and outputs, for which we can write a determinism specification. Rather, parallelism correctness for such libraries means that the libraries behave "correctly" when a parallel client makes concurrent calls into the library.

In both cases above, the key is the possibility of parallel *interference*. In our branch-and-bound example, multiple iterations of the `coforeach`-loop may concurrently access and modify shared variables `queue`, `lowest_cost`, and `min_cost_soln`. Is our example written such that each loop iteration behaves correctly for any possible concurrent accesses by other iterations, or can there be harmful interference that leads to an incorrect program result?

```
 1:   coforeach (work in queue) {
          @assert_atomic {
 2:           if (lower_bound_cost(work) >= lowest_cost)
 3:               continue;
 4:           if (size(work) < THRESHOLD) {
 5:               soln = find_min_cost_solution(work);
 6:               synchronized (lock) {
 7:                   if (cost(soln) < lowest_cost) {
 8:                       lowest_cost = cost(soln);
 9:                       min_cost_soln = soln;
10:                   }
11:               }
12:           } else {
13:               queue.add(split(work))
14:           }
          }
15:   }
             semantically atomic with respect to:
          (lowest_cost == lowest_cost')
              && min_cost_soln.equals(min_cost_soln')
```

Figure 2.2: Our running example, with a semantic atomicity specification.

A fundamental non-interference property for parallel programs is *atomicity* [64]. A block of code is said to be *atomic* if it appears to execute all at once, indivisibly and without interruption from any other program thread. Thus, in Chapter 5, we propose *semantic atomicity* specifications, to allow programmers to specify that certain program regions are, at a high-level, free from parallel and concurrent interference.

In Figure 2.2, we give a semantic atomicity specification for our example program. We have enclosed the body of the `coforeach`-loop in an `@assert_atomic` specification block. This specification requires that, for any parallel execution of the branch-and-bound search, producing `lowest_cost` and `min_cost_soln`, there must exist an *equivalent*, *serial* execution:

- [**Equivalent**] This other execution must produce `lowest_cost'` and `min_cost_soln'` that are semantically equivalent according to the given bridge predicate:

  ```
  lowest_cost==lowest_cost' && min_cost_soln.equals(min_cost_soln')
  ```

- [**Serial**] In this other execution, each `@assert_atomic` block executes all-at-once and indivisibly — i.e., once some thread begins executing an `@assert_atomic` block, no other thread may execute any instructions until the first thread has finished executing the entire `@assert_atomic` block.

If our specification is satisfied, it means that, through the use of locking and a thread-safe `queue`, we have successfully implemented the body of the `coforeach` loop so that it executes as if it were atomic. That is, although multiple loop iterations may execute in parallel, interleaving their instructions and concurrently accessing `lowest_cost`, `min_cost_soln`, and `queue`, we have used sufficient concurrency control to ensure that the final result is always a result we *could have gotten* if each loop iteration were actually executed all-at-once, without interruption by any other thread.

Unlike in our determinism specification in Section 2.3, in our semantic atomicity specification we can say something about parallelism correctness with respect to `min_cost_soln`. Recall that a determinism specification requires that, for any parallel execution, *all* other parallel executions produce equivalent results. Thus, we could not say anything about `min_cost_soln`, because our branch-and-bound search can correctly return different minimum-cost solutions in different executions on the same input. But our atomicity specification only requires that, for any parallel execution, *there exists one* execution that produces equivalent results, although this execution must be *serial*. This difference allows us capture the parallelism correctness of the nondeterministic aspects of our running example.

This difference also means, however, that we cannot capture desired *determinism* with atomicity specifications. In particular, unlike our determinism spec, our semantic atomicity spec in Figure 2.2 does not require that every parallel thread schedule return the same `lowest_cost`. Thus, our atomicity and determinism specifications complement each other.

Note that, as in our determinism specifications, the use of bridge predicates allows us to specify atomicity with respect *semantic* equivalence, at the level of abstraction of our application. We show in Chapter 5 that focusing on such high-level atomicity allows us to write useful specifications for many more programs than if we used low-level bit-by-bit equivalence typically used in traditional atomicity work.

Testing Semantic Atomicity Specifications

Suppose that we want to test that our example program conforms to its semantic atomicity specification in Figure 2.2. That is, given an execution of our parallel branch-and-bound search, producing `lowest_cost` and `min_cost_soln`, we want to check if any *serial* executions produce equivalent final results. It is clear that, for most programs, we cannot perform such a general check — there will be far too many possible serial executions for us to enumerate all of them in order to compare their results to those of our original parallel execution.

In Chapter 5, we show that, with two key insights, we can effectively test semantic atomicity specifications. First, we show that, for a given parallel execution, it is sufficient to consider only *strict serializations* [115] of the execution when searching for equivalent, serial behavior. For example, consider the parallel execution described in Figure 2.3 on the next page. In this execution, two pairs of `@assert_atomic` blocks execute concurrently, those for loop iterations **B** and **C** and those for loop iterations **E** and **F**. In searching for equivalent serial executions, we will examine only:

- Serial executions which run the exact same set of `@assert_atomic` blocks. Such executions are *serializations* [115] of the original parallel execution, and with this restriction we could be said to be checking the *serializability* of our specified atomic blocks.

  Thus, for example, we will not consider the serial execution described by Figure 2.4 on the next page. In this trace, because of differences in execution order[1], different regions of the solution space are pruned and a different set of `@assert_atomic` blocks are run (i.e., on different elements `work` of `queue`).

- Serial executions which preserve the order of `@assert_atomic` blocks that did not overlap in the parallel execution. Such executions are *strict serializations* [115], or *linearizations* [100], of the original execution, and with this additional restriction we could be said to be checking the *strict serializability*, or *linearizability*, of our specified atomic blocks.

  Thus, for example, we will not consider the serial execution described in Figure 2.5, as atomic blocks **D**, **E**, and **F** are all executed *before* atomic block **C**, whereas block **C** was executed entirely before blocks **D**, **E**, and **F** in the original parallel execution.

Even restricted to strict serializations, however, we could have far too many serial executions to examine in checking the semantic atomicity of a given parallel execution. Our second insight in Chapter 5 is that we can effectively test a program's semantic atomicity by only testing the strict serializability of parallel executions in which a small number of atomic blocks *overlap* (i.e. execute concurrently). For example, in the parallel execution in Figure 2.3, there are only two pairs of overlapping dynamic instances of atomic blocks:

[1]Note that elements of `queue` can be processed out-of-order in a serial execution because, although loop iterations are started for each element of the queue in order, a later loop iteration can begin its `@assert_atomic` block before an earlier parallel iteration.

B with **C**, and **E** with **F**. Thus, as the relative orderings of all other pairs of atomic blocks are fixed, there are at most *four* possible strictly-serialized executions that we must examine in our atomicity testing for this parallel execution. One such serial execution is shown in Figure 2.6 — in this execution, the `@assert_atomic` block **B** is executed before block **C**, and block **F** is executed before block **E**.

In Chapter 5, we propose a novel testing technique that: (1) generates parallel executions with a small number of overlapping `@assert_atomic` blocks, and (2) enumerates all linearizations of the each generated parallel execution in order to test semantic atomicity. In Chapter 5, we apply our testing technique to a number of parallel and concurrent Java benchmarks and find several previously-unknown atomicity errors, including in the widely-used `java.util.concurrent` library.

Note that, as with our determinism checking, we are testing *semantic* atomicity, comparing only final program results and comparing those results with high-level bridge predicates. As we discuss in Chapter 5, this is quite different than traditional approaches to atomicity checking – in particular, *conflict-serializability* [115] checking [62], which imposes strict conditions on all of the low-level shared reads and writes in a program. We show in Chapter 5 that our higher-level approach allows to specify and test the atomicity of a much wider range of programs.

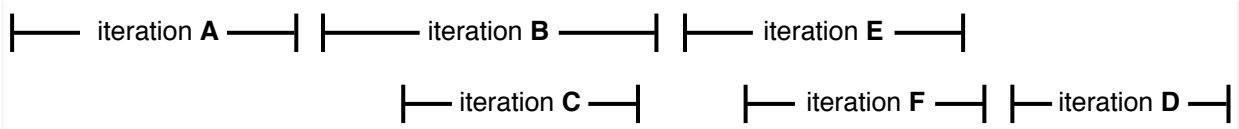

Figure 2.3: A trace of a parallel execution of our running example from Figure 2.2, showing when each `@assert_atomic` block started and ended. The execution is shown left-to-right — blocks that executed concurrently are shown overlapping vertically. Each block is identified by the `work` item processed by its loop iteration (**A**, **B**, ...).

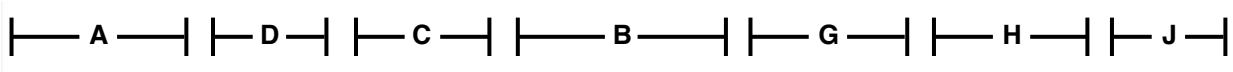

Figure 2.4: A trace of a *serial* execution of our example program.

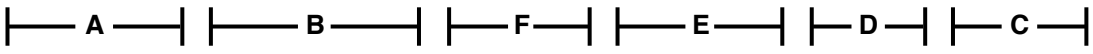

Figure 2.5: A serial execution that is a *serialization* of the parallel trace from Figure 2.3.

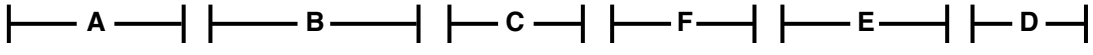

Figure 2.6: A serial execution that is a *strict serialization* of the trace from Figure 2.3.

2.5 Nondeterministic Sequential Specifications

In Sections 2.3 and 2.4, we have shown that semantic determinism specifications and semantic atomicity specifications both allow us to specify critical, high-level parallelism correctness properties for our example parallel branch-and-bound program. But, these two specifications are only *partial* specifications of parallelism correctness.

That is, imagine we had a formal proof that our running example program satisfied both our semantic determinism specification from Section 2.3 and our semantic atomicity specification from Figure 2.2. Is our example program definitely free from any errors in its use of parallelism? Can we turn our attention to the functional correctness of the program — i.e., whether or not it produce a correct output, given its input — knowing that we are done addressing the parallelism correctness? And can we test, debug, and verify the functional correctness *in a sequential way*, independent of the interleaving of parallel threads?

Unfortunately, the answer to the above questions is no. As we showed in the previous two sections, our determinism specification tells us nothing about the parallelism correctness with respect to the final `min_cost_soln` found. And, even knowing that every `@assert_atomic` block executes *as if* actually atomic, we are still left with a program with parallel threads that interleave nondeterministically.

In Chapter 6, we propose *nondeterministic sequential* (NDSeq) specifications as a complete specification for parallelism correctness for programs with structured parallelism (e.g, parallel `coforeach` loops or `cobegin` blocks). The key idea is to specify the parallelism correctness of a parallel program using a version of the program that is *sequential*, but also has some controlled *nondeterminism*. We say that a program's use of parallelism is correct if, for every execution of the program, there is an execution of the NDSeq specification program that produces an equivalent result. Thus, if a program conforms to its NDSeq spec, then we can test, debug, verify, etc., the program's functional correctness on the nondeterministic, sequential version of the program, without having to deal with parallel threads.

In Figure 2.7 on the next page, we give an NDSeq specification for our branch-and-bound example program. The specification program is very similar to the original parallel program, but the parallel `coforeach` loop has been replaced with a sequential `nd-foreach` loop. That is, in the NDSeq specification, there are no parallel threads and only one loop iteration ever runs at a time. In the NDSeq specification, we have replaced the uncontrolled nondeterminism of parallel interleavings with two controlled, nondeterministic constructs:

- The `nd-foreach` loop at Line 1. Although the `nd-foreach` loop is sequential, unlike a normal `foreach` loop, the `nd-foreach` can operate on the elements in `queue` in any nondeterministic order. At the beginning of a loop iteration, the `nd-foreach` loop can set `work` to *any* element chosen nondeterministically and removed from `queue`.
- The pruning check at Line 2. The star ("`*`") added to the condition can evaluate nondeterministically to `true` or `false`. Thus, when the NDSeq specification program compares `lower_bound_cost(work)` to `lowest_cost`, even when the lower bound exceeds the lowest cost found so far, the NDSeq specification can nondeterministically choose *not to prune*, instead executing the rest of loop iteration from Line 4 onward.

```
 1:   nd-foreach (work in queue) {
 2:       if (* && lower_bound_cost(work) >= lowest_cost)
 3:           continue;
 4:       if (size(work) < THRESHOLD) {
 5:           soln = find_min_cost_solution(work);
 6:           synchronized (lock) {
 7:               if (cost(soln) < lowest_cost) {
 8:                   lowest_cost = cost(soln);
 9:                   min_cost_soln = soln;
10:               }
11:           }
12:       } else {
13:           queue.add(split(work))
14:       }
15:   }
                equivalence is with respect to:
          (lowest_cost == lowest_cost')
              && min_cost_soln.equals(min_cost_soln')
```

Figure 2.7: A nondeterministic sequential (NDSeq) specification for our running example.

We now say that our example program's use of parallelism is correct if, for every execution of our parallel branch-and-bound search, *there exists one* execution of the NDSeq specification program that produces equivalent results. As with our determinism and atomicity specifications, we can use a bridge predicate to specify what it means for program results to be equivalent — i.e. identical `lowest_cost` and semantically equal `min_cost_soln`.

At first, NDSeq specifications may appear to be a restriction of our semantic atomicity specifications, where the entire body of each thread must be enclosed in a specified atomic block — so that a *serial* execution is, in fact, a *sequential* execution, with no interleaving of threads. But NDSeq specifications are also a generalization of semantic atomicity specifications, because the NDSeq specification contains nondeterministic constructs (i.e., `nd-foreach` and "`*`") not present in the original, parallel program.

We think of an NDSeq specification as indicating all of the nondeterminism a programmer expects in his or her parallel application — if the interleaving of parallel threads causes any additional, unintended nondeterminism, leading to non-equivalent final results, then there is a parallel error. For our branch-and-bound example, we specify that the nondeterminism of thread scheduling should have no more effect on the final results than: (1) processing items from the `queue` in a nondeterministic order, and (2) nondeterministically not pruning some `work` items could have been pruned.

In Chapter 6, we formally describe our proposed NDSeq specifications. We also identify several common patterns for writing NDSeq specs. Using these patterns, we were able to write NDSeq specs for the complete parallelism correctness of a number of Java benchmarks, even in cases where the functional correctness was far too complex for us to specify.

Testing NDSeq Specifications

Suppose that we want to test that our example program conforms to its nondeterministic sequential specification in Figure 2.7. That is, given an execution of our parallel branch-and-bound search, producing `lowest_cost` and `min_cost_soln`, we want to check if any executions of our NDSeq specification produce equivalent final results.

We observe that this checking problem appears quite similar to the problem of checking semantic atomicity, described in Section 2.4. Indeed, an NDSeq specification is like an atomicity specification with added nondeterminism, except that the entire body of every parallel thread must be declared to be atomic — e.g., in our example program, our NDSeq specification, in essence, is entirely enclosing each loop iteration in an atomic block, including the removal of an element from the `queue`.

The atomicity testing approach we described in Section 2.4, however, cannot easily be applied to testing NDSeq specifications. Even if we could use a similar approach to limit the number of orderings of `nd-foreach` loop iterations that we had to examine, we would have far too many executions to enumerate because, in searching for an NDSeq execution producing equivalent final results, we would have to try out all possible evaluations of the nondeterministic star ("*") construct at Line 2.

Rather, we attack the problem of testing against NDSeq specifications by using a technique from traditional atomicity testing [62]: *conflict-serializability* [115]. In traditional atomicity checking, one constructs a *conflict graph*, whose vertices are the dynamic atomic blocks (often called "transactions") run in a given parallel execution. One adds an edge from one dynamic instance of any atomic block to another whenever:

- The first block accesses some shared variable v.
- The second block accesses the same shared variable v later in the execution.
- At least one of the accesses is a write.

In traditional atomicity checking, one further adds an edge from earlier atomic blocks to later atomic blocks in the same thread. A classic theorem from the theory of database concurrency control [115] states that, if there are no cycles in this conflict graph, then the atomic blocks in the execution are *serializable*. In fact, the theorem shows that the parallel execution can be rearranged — by swapping adjacent, non-conflicting instructions from different threads — into an execution that behaves identically, but in which every atomic block is executed all-at-once, without interruption by any other thread.

We show in Chapters 5 and 6 that conflict-serializability, on its own, is too strict for testing the parallelism correctness of many real applications. But, in Chapter 6 we propose a way to generalize conflict-serializability checking in order to effectively check parallel programs against their NDSeq specifications. The key idea is that, by combining the nondeterminism added to the NDSeq specification (i.e, the "*" construct) with a dynamic dependence analysis, we can show that certain conflict accesses are *irrelevant* and can be safely ignored during conflict-serializability checking.

For example, consider an execution of our parallel branch-and-bound procedure in which:

R1. First, parallel loop iteration t_1 reads shared variable `lowest_cost` at Line 2, and does not prune its region of the solution space.

W2. Then, a different parallel loop iteration t_2 writes to shared variable `lowest_cost`.

R3. Finally, iteration t_1 reads `lowest_cost` again, this time at Line 7.

In traditional conflict-serializability checking, we see a conflict edge from t_1 to t_2 — because t_1 reads `lowest_cost` (R1) before t_2 writes to it (W2) — and a conflict edge from t_2 to t_1 — because t_2 writes to `lowest_cost` (W2) before t_1 reads it (R3). Thus, we would declare the execution non-serializable and a potential parallelism error.

But with our generalized conflict-serializability checking, we observe that:

1. The conflicting read (R1) of shared variable `lowest_cost` is only used once — in comparing `lowest_cost` to `lower_bound_cost(work)`.
2. Further, because `lowest_cost` < `lower_bound_cost(work)`, iteration t_1 did *not* prune. As a result, Lines 2–3 have no persisting local or global side effects.
3. Thus, in the NDSeq specification we can get the exact same behavior of Lines 2–3 — i.e., not pruning and having no side effects — if the "`*`" in the condition at Line 2 evaluates to `false`.

   Thus, any conflicts involving the read R1 of `lowest_cost` at Line 2 are *irrelevant*. In rearranging the instructions of our parallel execution in order to form an equivalent NDSeq execution, it is fine to swap the read R1 to after the write W2, because the resulting NDSeq execution can reproduce the same behavior — i.e., not pruning in iteration t_1 — by evaluating its "`*`" as `false` and then not even reading `lowest_cost`.

We prove in Chapter 6 that, if our checking technique can show that, after removing all irrelevant conflicts, no cycles remain in the conflict graph, then the observed parallel execution does conform to the NDSeq specification.

Unlike our technique from Section 2.4 for testing semantic atomicity, our generalized conflict-serializability checking for NDSeq specifications requires no enumeration of nondeterministic, sequential executions. Instead, we only need to determine which reads and writes are relevant, via our dynamic dependence analysis, and then check for conflict-cycles among the remaining accesses. One trade-off when using a technique focusing on low-level reads and writes, however, is that we can no longer use any bridge predicates for specifying equivalence in our NDSeq specifications. Rather, we specify the set of variables that hold the final output (called *focus* variables), and our dynamic NDSeq checking analysis ensure that all writes to the focus variables are preserved. Another trade-off is that our NDSeq checking may give false warnings for parallel executions that are correct but not conflict-serializable, even with our dynamic dependence analysis.

In Chapter 6, we find that our runtime checking technique is effective in testing our NDSeq specifications for Java programs, eliminating nearly all of the false warnings traditional conflict-serializability would report.

Automatically Inferring NDSeq Specifications

In writing an NDSeq specification for a structured-parallel application, a programmer must: (1) Identify the program's *focus variables*, which hold the final result of the program, and (2) Add nondeterministic star ("*") constructs to some conditionals to specify the intended and expected nondeterministic behavior[2]. For example, to write our NDSeq specification in Figure 2.7 for our branch-and-bound application, we had to (1) identify `lowest_cost` and `min_cost_soln` as the focus variables, and (2) add a "*" to the conditional at Line 2. For our benchmarks in Chapters 6 and 7, we found it fairly simple to identify the focus variables, but often complex and time-consuming to identify every location where a nondeterministic "*" had to be added in order to capture all of a program's intended nondeterminism.

We could save programmer effort if we had a way to automatically complete this second task of identifying all of the necessary nondeterminism in an NDSeq specification. We must be careful, however, to not add too much nondeterminism, as we want an NDSeq specification that captures all of the nondeterministic behavior of the corresponding parallel program, but we do not want to add any additional, unnecessary nondeterminism, which could complicate our efforts to sequentially test, debug, and verify functional correctness on the NDSeq specification.

In Chapter 7, we show that it is possible to infer likely placements of "*" constructs for an NDSeq specification for a parallel program, given a handful of representative executions of the program and a specification of the focus variables. The key idea is to find a minimum number of "*" that need to be added to the NDSeq specification so that the observed parallel executions all conform to the specification. We find such "*" and their placement in the NDSeq specification by encoding the specification inference problem as a minimum-cost satisfiability instance (MinCostSAT). In particular, given a set of parallel executions, we construct a SAT instance that corresponds to running our dynamic NDSeq checking algorithm from Section 2.5 on each execution. That is, there are constraints corresponding to running the dynamic dependence analysis to identify relevant and irrelevant reads and writes, as well constraints to check for cycles in the conflict graph, and the constraints are solvable only if there exists an NDSeq specification (i.e., "*" placements) to which all of the observed executions conform. There is a variable in our instance for each location to which a nondeterministic "*" construct could be added, so we can use a MinCostSAT solver to find a solution in which a minimum number of "*" are added.

We implement nDetermin, our specification inference algorithm, in Chapter 7 and prove it to be sound, as well as complete, relative to our NDSeq checking algorithm. We apply nDetermin to the benchmarks from Chapter 6, and we find that nDetermin is able to automatically infer, for nearly all of our benchmarks, equivalent or better NDSeq specifications than those we previously wrote by hand. Further, for the two benchmarks with parallelism errors, nDetermin correctly reports that no NDSeq specification exists that either benchmark would satisfy.

[2]We automatically treat structured-parallelism constructs — e.g., `coforeach` — as nondeterministic sequential constructs in the NDSeq specification — e.g., `nd-foreach`.

Chapter 3

Asserting and Checking Determinism for Multithreaded Programs

Despite the increasing need for programmers to write parallel software to take advantage of multicore processors, parallel software remains more difficult to write and debug than its sequential counterpart. A key reason for this difficulty is that parallel programs can show different behaviors depending on how the executions of their parallel threads interleave.

The fact that executions of parallel threads can interleave with each other in arbitrary fashion to produce different outputs is called *internal nondeterminism* or *scheduler nondeterminism*. Internal nondeterminism is essential to make parallel threads execute simultaneously and to harness the power of parallel chips. However, most of the sequential programs that we write are *deterministic* — they produce the same output on the same input. Therefore, in order to make parallel programs easy to understand and debug, we need to make them behave like sequential programs, i.e. we need to make parallel programs deterministic.

A number of ongoing research efforts aim to make parallel programs deterministic by construction. These efforts include the design of new parallel programming paradigms [143, 96, 83, 4, 92, 16, 82] and the design of new type systems and annotations that could retrofit existing parallel languages [8, 20]. But such efforts face two key challenges. First, new languages see slow adoption and often remain specific to limited domains. Second, new paradigms often include restrictions that can hinder general purpose programming. For example, a key problem with new type systems is that they can make programming more difficult and restrictive.

Under the most widespread method for writing parallel programs, threads [80, 19, 41, 81], ensuring correct deterministic behavior can be very challenging. To aid programmers in writing deterministic programs, a number of tools and techniques have been developed. These tools attempt to automatically find sources of nondeterminism likely to be harmful (i.e. to lead to nondeterministic output) such as data races and high-level race conditions.

 A version of this chapter was published in the Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09), http://doi.acm.org/10.1145/1595696.1595700

A large body of work spanning over 30 years has focused on data race detection. A data race occurs when two threads concurrently access a memory location and at least one of the accesses is a write. Both dynamic [44, 2, 130, 122, 38, 3, 132] and static [139, 59, 23, 54, 71, 49, 120, 107] techniques have been developed to detect and predict data races in multithreaded programs. Although the work on data race detection has significantly helped in finding determinism bugs in parallel programs, it has been observed that (1) the absence of data races is not sufficient to ensure determinism [9, 65, 57], and (2) data races do not always cause nondeterministic results. In fact, race conditions are often useful in gaining performance, while still ensuring high-level deterministic behavior [14].

We argue that programmers should be provided with a framework that will allow them *to express deterministic behaviors of parallel programs directly and easily.* Specifically, we should provide an assertion framework where programmers can *directly and precisely* express the necessary deterministic behavior. On the other hand, the framework should be flexible enough so that deterministic behaviors can be expressed more *easily* than with a traditional assertion framework. For example, when expressing the deterministic behavior of a parallel edge detection algorithm for images, we should not have to rephrase the problem as a race detection problem; neither should we have to write a state assertion that relates the output to the input, which would be complex and time-consuming. Rather, we should simply be able to say that, if the program is executed on the same image, then the output image remains the same regardless of how the program's parallel threads are scheduled.

In this chapter, we propose such a framework for asserting that blocks of parallel code behave deterministically. Formally, our framework allows a programmer to give a specification for a block P of parallel code as:

```
deterministic assume(Pre(s_0, s'_0)) {
    P
} assert(Post(s, s'));
```

This specification asserts the following: Suppose P is executed twice with potentially different schedules, once from initial state s_0 and once from s'_0 and yielding final states s_1 and s'_1, respectively. Then, if the user-specified *precondition Pre* holds over s_0 and s'_0, then s and s' must satisfy the user-specified *postcondition Post*.

For example, we could specify the deterministic behavior of a parallel matrix multiply with the following:

```
deterministic assume(|A − A'| < 10^−6 and |B − B'| < 10^−6) {
    C = parallel_matrix_multiply_float(A, B);
} assert(|C − C'| < 10^−6);
```

Note the use of primed variables A', B', and C' in the above example. These variables represent the state of the matrices A, B, and C from a different execution. As such the predicates that we write inside `assume` and `assert` are different from state predicates written in a traditional assertion framework — these special predicates relate a pair of states from different executions. We call such a predicate a *bridge predicate* and an assertion using bridge

predicates a *bridge assertion*. A key contribution of this work is the introduction of these bridge predicates and bridge assertions. We believe that these novel predicates can be used not only for determinism specification, but also be used for other purposes such as writing regression tests.

Our determinism assertions provide a way to specify the correctness of the parallelism in a program independently of the program's traditional functional correctness. By checking whether different program schedules can nondeterministically lead to semantically different answers, we can find bugs in a program's use of parallelism even when unable to directly check functional correctness — i.e. that the program's output is correct given its input. Inversely, by checking that a parallel program behaves deterministically, we can gain confidence in the correctness of its use of parallelism independently of whatever method we use to gain confidence in the program's functional correctness.

We have implemented our determinism assertions as a library for the Java programming language. We evaluated the utility of these assertions by manually adding determinism specifications to a number of parallel Java benchmarks. We used an existing tool to find executions exhibiting data and higher-level races in these benchmarks and used our determinism assertions to distinguish between harmful and benign races. We found it to be fairly easy to specify the correct deterministic behavior of the benchmark programs using our assertions, despite being unable in most cases to write traditional invariants or functional correctness assertions. Further, our determinism assertions successfully distinguished the two known harmful races in the benchmarks from the benign races.

3.1 Determinism Specification

In this section, we motivate and define our proposal for assertions for specifying determinism.

Strictly speaking, a block of parallel code is said to be deterministic if, given any particular initial state, all executions of the code from the initial state produce the exact same final state. In our specification framework, the programmer can specify that they expect a block of parallel code, say P, to be deterministic with the following construct:

```
deterministic {
    P
}
```

This assertion specifies that if s and s' are both program states resulting from executing P under different thread schedules from some initial state s_0, then s and s' must be equal. For example, the specification:

```
deterministic {
    C = parallel_matrix_multiply_int(A, B);
}
```

asserts that for the parallel implementation of matrix multiplication (defined by function `parallel_matrix_multiply_int`), any two executions from the same program state must

reach the same program state — i.e. with identical entries in matrix `C` — no matter how the parallel threads are scheduled.

A key implication of knowing that a block of parallel code is deterministic is that we may be able to treat the block as sequential in other contexts. That is, although the block may have internal parallelism, a programmer (or perhaps a tool) can hopefully ignore this parallelism when considering the larger program using the code block. For example, perhaps a deterministic block of parallel code in a function can be treated as if it were a sequential implementation when reasoning about the correctness of code calling the function.

Semantic Determinism

The above determinism specification is often too conservative. For example, consider a similar example, but where `A`, `B`, and `C` are floating-point matrices:

```
deterministic {
    C = parallel_matrix_multiply_float(A, B);
}
```

Limited-precision floating-point addition and multiplication are not associative due to rounding error. Thus, depending on the implementation, it may be unavoidable that the entries of matrix `C` will differ slightly depending on the thread schedule.

In order to tolerate such differences, we must relax the determinism specification:

```
deterministic {
    C = parallel_matrix_multiply_float(A, B);
} assert(|C − C′| < 10^−6);
```

This assertion specifies that, for any two matrices C and C' resulting from the execution of the matrix multiply from same initial state, the entries of C and C' must differ by only a small quantity (i.e. 10^{-6}).

Note that the above specification contains a predicate over two states — each from a different parallel execution of deterministic block. We call such a predicate a *bridge predicate*, and an assertion using a bridge predicate a *bridge assertion*. Bridge assertions are different from traditional assertions in that they allow one to write a property over two program states coming from different executions whereas traditional assertions only allow us to write a property over a single program state.

Note also that such predicates need not be equivalence relations on pairs of states. In particular, the approximate equality used above is not an equivalence relation.

This relaxed notion of determinism is useful in many contexts. Consider the following example that, in parallel, adds two items to a synchronized set:

```
Set set = new SynchronizedTreeSet();
deterministic {
    cobegin {
        set.add(3);
        set.add(5);
    }
} assert(set.equals(set'));
```

If, for example, `set` is represented internally as a red-black tree, then a strict determinism assertion would be too conservative. The structure of the resulting tree, and its layout in memory, will almost certainly differ depending on which element is inserted first, and thus the different executions can yield different program states.

But we can use a bridge predicate to assert that, no matter what schedule is taken, the resulting set is *semantically* equal. That is, for objects *set* and *set'* computed by two different schedules, the `equals` method must return true because the sets must logically contain the same elements. We call this *semantic determinism*.

Preconditions for Determinism

So far we have described the following construct:

```
deterministic {
    P
} assert(Post);
```

where *Post* is a predicate over two program states in different executions resulting from different thread schedules[1]. That is, if s and s' are two states resulting from any two executions of P from the same initial state, then $Post(s, s')$ holds.

The above construct could be rewritten in the following way:

```
deterministic assume(s0 = s0') {
    P
} assert(Post);
```

That is, if any two executions of P start from initial states s_0 and s'_0, respectively, and if s and s' are the resultant final states, then $s_0 = s'_0$ implies that $Post(s, s')$ holds. The above rewritten specification suggests that we can further relax the requirement of $s_0 = s'_0$ by replacing it with a bridge predicate $Pre(s_0, s'_0)$. For example:

[1]Note that in the above construct we do not refer to the final states s and s', but we make them implicit by assuming that *Post* maps a pair of program states to a Boolean value.

```
deterministic assume(set.equals(set')) {
    cobegin {
        set.add(3);
        set.add(5);
    }
} assert(set.equals(set'));
```

The above specification states that if any two executions start from sets containing the same elements, then after the execution of the code, the resulting sets after the two executions must still contain exactly the same elements.

Comparison to Traditional Assertions

In summary, we propose the following construct for specifying deterministic behavior:

```
deterministic assume(Pre) {
    P
} assert(Post);
```

Formally, it states that for any two program states s_0 and s'_0, whenever:

- $Pre(s_0, s'_0)$ holds,
- an execution of P from s_0 terminates and results in state s,
- and an execution of P from s'_0 terminates and results in state s',

then $Post(s, s')$ must hold.

Note that the use of bridge predicates *Pre* and *Post* has the same flavor as pre and postconditions used for functions in program verification. However, unlike traditional pre and postconditions, the proposed *Pre* and *Post* predicates relate pairs of states from two different executions. In traditional verification, a precondition is usually written as a predicate over a single program state, and a postcondition is usually written over two states — the states at the beginning and end of the function. For example:

```
foo() {
    assume(x > 0);
    old_x = x;
    x = x * x;
    assert(x == old_x*old_x);
}
```

The key difference between a traditional postcondition and a *Post* predicate is that a postcondition relates two states at different times along a same execution, whereas a *Post* predicate relates two program states in different executions.

Advantages of Determinism Assertions

Our determinism specifications are a middle ground between the implicit specification used in race detection — that programs should be free of data races — and the full specification of functional correctness. It is a great feature of data race detectors that typically no programmer specification is needed. However, manually determining which reported races are benign and which are bugs can be time-consuming and difficult. We believe our determinism assertions, while requiring little effort to write, can greatly aid in distinguishing harmful from benign data races (or higher-level races).

One could argue that a determinism specification framework is unnecessary given that we can write the functional correctness of a block of code using traditional pre- and postconditions. For example, one could write the following to specify the correct behavior of `parallel_matrix_multiply_float`:

```
C = parallel_matrix_multiply_float(A, B);
assert(|C − A × B| < 10^−6);
```

We agree that, if one can write such a functional specification, then there is no need to write determinism specification, as functional correctness implies deterministic behavior.

The advantage of our determinism assertions, however, are that they provide a way to specify the correctness of just the use of parallelism in a program, independent of the program's full functional correctness. In many situations, writing a full specification of functional correctness is difficult and time consuming. But, a simple determinism specification enables us to use automated technique to check for parallelism bugs, such as harmful data races causing semantically nondeterministic behavior.

Consider a parallel function `parallel_edge_detection` that takes an image as input and returns an image where detected edges have been marked. Relating the output to the input image with traditional pre- and postconditions would likely be quite challenging. However, it is simple to specify that the routine does not have any parallelism bugs causing a correct image to be returned for some thread schedules and an incorrect image for others:

```
deterministic assume(img.equals(img')) {
    result = parallel_edge_detection(img);
} assert(result.equals(result'));
```

where `img.equals(img')` returns true iff the two images are pixel-by-pixel equal.

For this example, a programmer could gain some confidence in the correctness of the routine by writing unit tests or manually examining the output for a handful of images. He or she could then use automated testing or model checking to separately check that the parallel routine behaves deterministically on a variety of inputs, gaining confidence that the code is free from concurrency bugs.

We believe that it is often difficult to come up with effective functional correctness assertions. However, it is often quite easy to use bridge assertions to specify deterministic behavior, enabling a programmer to check for harmful concurrency bugs. In the Evaluation section, we provide several case studies to support this argument.

3.2 Checking Determinism

There may be many potential approaches to checking or verifying a determinism specification, from testing to model checking to automated theorem proving. In this section, we propose a simple and incomplete method for checking determinism specifications at run-time.

The key idea of the method is that, whenever a deterministic block is encountered at run-time, we can record the program states s_{pre} and s_{post} at the beginning and end of the block. Then, given a collection of $(s_{\text{pre}}, s_{\text{post}})$ pairs for a particular deterministic block in some program, we can check a determinism specification, albeit incompletely, by comparing pairwise the pairs of initial and final states for the block. That is, for a deterministic block:

```
deterministic assume(Pre) {
    P
} assert(Post);
```

with pre- and post-predicates *Pre* and *Post*, we check for every recorded pair of pairs $(s_{\text{pre}}, s_{\text{post}})$ and $(s'_{\text{pre}}, s'_{\text{post}})$ that:

$$Pre(s_{\text{pre}}, s'_{\text{pre}}) \implies Post(s_{\text{post}}, s'_{\text{post}})$$

If this condition does not hold for some pair, then we report a determinism violation.

To increase the effectiveness of this checking, we must record pairs of initial and final states for deterministic blocks executed under a wide variety of possible thread interleavings. Thus, in practice we likely want to combine our determinism assertion checking with existing techniques and tools for exploring parallel schedules of a program, such as noise making [45, 140], active random scheduling [131, 132], or model checking [156].

In practice, the cost of recording and storing entire program states could be prohibitive. However, real determinism predicates often depend on just a small portion of the whole program state. Thus, we need only to record and store small projections of program states. For example, for a determinism specification with pre- and post-predicate `set.equals(set')` we need only to save object `set` and its elements (possibly also the memory reachable from these objects), rather than the entire program memory. This storage cost sometimes can be further reduced by storing and comparing checksums or approximate hashes.

3.3 Determinism Checking Library

In this section, we describe the design and implementation of an assertion library for specifying and checking determinism of Java programs.

Note that, while it might be preferable to introduce a new syntactic construct for specifying determinism, we instead provide the functionality as a library for simplicity of the implementation.

Figure 3.1 shows the core API for our determinism assertion library. Functions `open` and `close` specify the beginning and end of a deterministic block. Deterministic blocks may be

```
class Deterministic {

    static void open()

    static void close()

    static void assume(Object o, Predicate p)

    static void assert(Object o, Predicate p)

    interface Predicate {
        boolean apply(Object a, Object b)
    }
}
```

Figure 3.1: Core determinism specification API.

nested, and each block may contain multiple calls to functions `assume` and `assert`, which are used to specify the pre- and post-predicates of deterministic behavior.

Each call `assume`(o, pre) in a deterministic block specifies part of the pre-predicate by giving some projection o of the program state and a predicate *pre*. That is, it specifies that one condition for any execution of the block to compute an equivalent, deterministic result is that *pre*.`apply`(o, o') return *true* for object o' from the other execution.

Similarly, a call `assert`$(o, post)$ in a deterministic block specifies that, for any execution satisfying every `assume`, predicate *post*.`apply`(o, o') must return *true* for object o' from the other execution.

At run-time, our library records every object (i.e. state projection) passed to each `assert` and `assume` in each deterministic block, saving them to a central, persistent store. We require that all objects passed as state projections implement the `Serializable` interface to facilitate this recording. (In practice, this does not seem to be a heavy burden. Most core objects in the Java standard library are serializable, including numbers, strings, arrays, lists, sets, and maps/hashtables.)

Then, also at run-time, a call to `assert`$(o, post)$ checks *post* on o and all o' saved from previous, matching executions of the same deterministic block. If the post-predicate does not hold for any of these executions, a determinism violation is immediately reported. Deterministic blocks can contain many `assert`'s so that determinism bugs can be caught as early as possible and can be more easily localized.

For flexibility, programmers are free to write state projections and predicates using the full Java language. However, it is a programmer's responsibility to ensure that these predicates contain no observable side effects, as there are no guarantees as to how many times such a predicate may be evaluated in any particular run.

So that the library is easy to use, it tracks which threads are in which deterministic blocks. Thus, a call to `assume`, `assert`, or `close` is automatically associated with the correct enclosing block, even when called from a spawned, child thread. The only restriction on the location of these calls is that every `assume` call in a deterministic block must occur before any `assert`.

Built-in Predicates

For programmer convenience, we provide two built-in predicates that are often sufficient for specifying pre- and post-predicates for determinism. The first, `Equals`, returns *true* if the given objects are equal using their built-in `equals` method — that is, if o.`equals`(o'). For many Java objects, this method checks semantic equality — e.g. for integers, floating-point numbers, strings, lists, sets, etc. Further, for single- or multi-dimensional arrays (which do not implement such an `equals` method), the `Equals` predicate compares corresponding elements using their `equals` methods. Figure 3.2 gives an example with `assert` and `assume` using this `Equals` predicate.

The second predicate, `ApproxEquals`, checks if two floating-point numbers, or the corresponding elements of two floating-point arrays, are within a given margin of each other. As shown in Figure 3.3, we found this predicate useful in specifying the deterministic behavior of numerical applications, where it is unavoidable that the low-order bits may vary with different thread interleavings.

Real-World Floating-Point Predicates

In practice, floating-point computations often have input-dependent error bounds. For example, we may expect any two runs of a parallel algorithm for summing inputs $x_1, \ldots, x_n$ to return answers equal to within $2N\epsilon \sum_i |x_i|$, where ϵ is the machine epsilon. We can assert:

```
sum = parallel_sum(x);
bound = 2 * x.length * ϵ * sum_of_abs(x);
Predicate apx = new ApproxEquals(bound);
Deterministic.assert(sum, apx);
```

As another example, different runs of a molecular dynamics simulation may be expected to produce particle positions equal to within something like ϵ multiplied by the sum of the absolute values of all initial positions. We can similarly compute this value at the beginning of the computation, and use an `ApproxEquals` predicate with the appropriate bound to compare particle positions.

Concrete Example: mandelbrot

Figure 3.2 shows the determinism assertions we added to one of our benchmarks, a program for rendering images of the Mandelbrot Set fractal from the Parallel Java Library [86].

The benchmark first reads a number of integer and floating-point parameters from the command-line. It then spawns several worker threads, which each compute the hues for different segments of the final image, storing them in shared array `matrix`. After waiting for all of the worker thread to finish, the program encodes and writes the image to a file given as a command-line argument.

To add determinism annotations to this program, we simply opened a deterministic block just before the worker threads are spawned and closed it just after they are joined. At the beginning of this block, we added an `assume` call for each of the seven fractal parameters, such as the image size and and color palette. At the end of the block, we assert that the resulting array `matrix` should be deterministic, however the worker threads are interleaved.

Note that it would be quite difficult to add assertions for the functional correctness of this benchmark, as each pixel of the resulting image is a complex function of the inputs (i.e. the rate at which a particular complex sequence diverges). Further, there do not seem to be any simple traditional invariants on the program state or outputs which would help identify a parallelism bug.

```
main(String args[]) {
    // Read parameters from command-line.
    ...

    // Pre-predicate: equal parameters.
    Predicate equals = new Equals();
    Deterministic.open();
    Deterministic.assume(width, equals);
    Deterministic.assume(height, equals);
    ...
    Deterministic.assume(gamma, equals);

    // spawn threads to compute fractal
    int matrix[][] = ...;
    ...

    // join threads
    ...

    Deterministic.assert(matrix, equals);
    Deterministic.close();

    // write fractal image to file
    ...
}
```

Figure 3.2: Determinism assertions for a Mandelbrot Set implementation from the Parallel Java Library [86].

```
main(String args[]) {
    // Read parameters from command-line.
    ...

    // Pre-predicate: equal parameters.
    Deterministic.open();
    Predicate equals = new Equals();
    Deterministic.assume(mm, equals);
    Deterministic.assume(PARTSIZE, equals);

    // spawn worker threads
    double ek[] = ..., epot[] = ..., vir[] = ...;
    ...

    // Deterministic final energies.
    Predicate apx = new ApproxEquals(1e-10);
    Deterministic.assert(ek[0], apx);
    Deterministic.assert(epot[0], apx);
    Deterministic.assert(vir[0], apx);
    Deterministic.close();
}

void run() {  // worker thread
    ... 100 lines of initialization ...
    particle[] particles = ...;
    double force[] = ...;

    for (int i = 0; i < num_iters; i++) {
        // update positions and velocities
        ...
        synchronizeBarrier()
        Predicate pae = new ParticleApproxEquals(1e-10);
        Deterministic.assert(particles, pae);
        synchronizeBarrier()

        // update forces
        ... 100 lines plus library calls ...
        synchronizeBarrier()
        Predicate apx = new ApproxEquals(1e-10);
        Deterministic.assert(force, apx);
        synchronizeBarrier()

        // temperature scale + sum energy
        ... 40 lines ...
        synchronizeBarrier();
        Deterministic.assert(ek, apx);
        Deterministic.assert(epot, apx);
        Deterministic.assert(vir, apx);
        synchronizeBarrier();
    }
}
```

Figure 3.3: Determinism assertions for `moldyn`, a molecular dynamics simulator from the Java Grande Forum Benchmark Suite [46].

Implementation

Due to the simple design, we were able to implement our determinism assertion library in only a few hundred lines of Java code. We use the Java `InheritableThreadLocal` class to track which threads are in which deterministic blocks (and so that spawned child threads inherit the enclosing deterministic block from their parent).

Currently, pairs of initial and final states for the deterministic blocks of an application are just recorded in a single file in the application's working directory. Blocks are uniquely identified by their location in an application's source (accessible through, e.g., a stack trace). When a determinism violation is detected, a message is printed and the application is halted.

3.4 Experimental Evaluation

In this section, we describe our efforts to validate two claims about our proposal for specifying and checking deterministic parallel program execution:

1. First, determinism specifications are easy to write. That is, even for programs for which it is difficult to specify traditional invariants or functional correctness, it is relatively easy for a programmer to add determinism assertions.

2. Second, determinism specifications are useful. When combined with tools for exploring multiple thread schedules, determinism assertions catch real parallelism bugs that lead to semantic nondeterminism. Further, for traditional concurrency issues such as data races, these assertions provide some ability to distinguish between benign cases and true bugs.

To evaluate these claims, we used a number of benchmark programs from the Java Grande Forum (JGF) benchmark suite [46], the Parallel Java (PJ) Library [86], and elsewhere. The names and sizes of these benchmarks are given in Table 3.1. The JGF benchmarks include five parallel computation kernels — for successive order-relaxation (`sor`), sparse matrix-vector multiplication (`sparsematmult`), coefficients of a Fourier series (`series`), cryptography (`crypt`), and LU factorization (`lufact`) — as well as a parallel molecular dynamic simulator (`moldyn`), ray tracer (`raytracer`), and Monte Carlo stock price simulator (`montecarlo`). Benchmark `tsp` is a parallel Traveling Salesman branch-and-bound search [122]. These benchmarks are standard, and have been used to evaluate many previous analyses for parallel programs (e.g. [111, 57, 132]). The PJ benchmarks include an app computing a Monte Carlo approximation of π (`pi`), a parallel cryptographic key cracking app (`keysearch3`), an app for parallel rendering Mandelbrot Set images (`mandelbrot`), and a parallel branch-and-bound search for optimal phylogenetic trees (`phylogeny`). Note that the benchmarks range from a few hundred to a few thousand lines of code, with the Parallel Java benchmarks relying on an additional 10-20,000 lines of library code from the Parallel Java Library (for threading, synchronization, and other functionality).

Benchmark		Approximate Lines of Code (App + Library)	Lines of Specification (+ Predicates)	Threads	Data Races		High-Level Races	
					Found	Determinism Violations	Found	Determinism Violations
JGF	`sor`	300	6	10	2	0	0	0
	`sparsematmult`	700	7	10	0	0	0	0
	`series`	800	4	10	0	0	0	0
	`crypt`	1100	5	10	0	0	0	0
	`moldyn`	1300	6	10	2	0	0	0
	`lufact`	1500	9	10	1	0	0	0
	`raytracer`	1900	4	10	3	**1**	0	0
	`montecarlo`	3600	4 + 34	10	1	0	2	0
PJ	`pi`	150 + 15,000	5	4	9	0	1+	**1**
	`keysearch3`	200 + 15,000	6	4	3	0	0+	0
	`mandelbrot`	250 + 15,000	10	4	9	0	0+	0
	`phylogeny`	4400 + 15,000	8	4	4	0	0+	0
	`tsp`	700	4	5	6	0	2	0

Table 3.1: Summary of experimental evaluation of determinism specifications. A single deterministic block specification was added to each benchmark. Each specification was checked on executions with races found by the CalFuzzer [132, 116, 84] tool.

Ease of Use

We evaluate the ease of use of our determinism specification by manually adding assertions to our benchmark programs. One deterministic block was added to each benchmark.

The third column of Table 3.1 records the number of lines of specification (and lines of custom predicate code) added to each benchmark. Overall, the specification burden is quite small. Indeed, for the majority of the programs, we were able to add determinism assertions in only five to ten minutes per benchmark, despite being unfamiliar with the code. In particular, it was typically not difficult to both identify regions of code performing parallel computation and to determine from documentation, comments, or source code which results were intended to be deterministic. Figures 3.2 and 3.3 show the (slightly cleaned up) assertions added to the `mandelbrot` and `moldyn` benchmarks.

The added assertions were correct on the first attempt for all but one benchmark. (For `phylogeny`, the resulting phylogenetic tree was erroneously specified as deterministic, when, in fact, there are many correct optimal trees. The specification was modified to assert only that the optimal score must be deterministic.)

The two predicates provided by our assertion library were sufficient for all but one of the benchmarks. For the JGF `montecarlo` benchmark, we had to write a custom `equals` and `hashCode` method for two classes — 34 total lines of code — in order to assume and assert that two sets, one of initial tasks and one of results, should be deterministic.

Further Determinism Assertions

Three of the benchmarks — `sor`, `moldyn`, and `lufact` — use barriers to synchronize their worker threads at many points during their parallel computations. These synchronization points provide locations where partial results of the computation can be specified to be deterministic. For example, as shown in Figure 3.3, we can assert in `moldyn` that the deterministic particle positions and forces should be computed in *every* iteration. Such intermediate assertions aid the early detection and localization of nondeterminism errors.

For these three benchmarks, we were able to add intermediate assertions at important synchronization barriers in only another fifteen to thirty minutes per benchmark. This adds roughly 25, 35, and 10 lines of specification, respectively, to `sor`, `moldyn`, `lufact`. Further, for the `moldyn` benchmark, this requires writing a custom predicate `ParticleApproxEquals` for comparing two arrays of `particle` objects for approximate equality of their positions and velocities, as well as customizing the serialization of `particle` objects.

Note, however, that care must be taken with such additional assertions to not capture an excessive amount of data. For example, it is not feasible to assert in every iteration of a parallel computation that a large intermediate matrix is deterministic — this requires serializing and checking a large enough quantity of data to have significant overhead. Checksums or approximate hashes, however, can greatly reduce this cost. In the above example, rather than storing and comparing the entire intermediate matrix, we can check only the row sums or the sum of all entries

Discussion

More experience, or possibly user studies, would be needed to conclude decisively that our assertions are easier to use than existing techniques for specifying that parallel code is correctly deterministic. However, we believe our experience is quite promising. In particular, writing assertions for the full functional correctness of the parallel regions of these programs seemed to be quite difficult, perhaps requiring implementing a sequential version of the code and asserting that it produces the same result. Further, there seemed to be no obvious simpler, traditional assertions that would aid in catching nondeterministic parallelism.

Despite these difficulties, we found that specifying the natural deterministic behavior of the benchmarks with our *bridge assertions* required little effort.

Effectiveness

To evaluate the utility of our determinism specifications in finding true parallelism bugs, we used a modified version of the CalFuzzer [132, 116, 84] tool to find real races in the benchmark programs, both data races and higher level races (such as races to acquire a lock). For each such race, we ran 10 trials using CalFuzzer to create real executions with these races and to randomly resolve the races (i.e. randomly pick a thread to "win"). We turned

on run-time checking of our determinism assertions for these trials, and recorded all found violations.

Table 3.1 summarizes the results of these experiments. For each benchmark, we indicate the number of real data races and higher-level races we observed. Further, we indicate how many of these races led to determinism violations in any execution.

In these experiments, the primary computational cost is from CalFuzzer, which typically has an overhead in the range of 2x-20x for these kinds of compute bound applications. We have not carefully measured the computational cost of our determinism assertion library. For most benchmarks, however, the cost of serializing and comparing a computation's inputs and outputs is dwarfed by the cost of the computation itself — e.g. consider the cost of checking that two fractal images are identical versus the cost of computing each fractal in the first place.

Determinism Violations

We found two cases of nondeterministic behavior. First, a known data race in the `raytracer` benchmark, due the use of the wrong lock to protect a shared sum, can cause an incorrect final answer to be computed.

Second, the `pi` benchmark can yield a nondeterministic answer given the same random seed because of insufficient synchronization of a shared random number generator. In each Monte Carlo sample, two successive calls to `java.util.Random.nextDouble()` are made. A context switch between these calls changes the set of samples generated. Similarly, `nextDouble()` itself makes two calls to `java.util.Random.next()`, which atomically generates up to 32 pseudo-random bits. A context switch between these two calls changes the generated sequence of pseudo-random doubles. Thus, although `Random.nextDouble()` is thread-safe and free of data races, scheduling nondeterminism can still lead to a nondeterministic result. (This behavior is known — the Parallel Java library provides several versions of this benchmark, one of which does guarantee a deterministic result for any given random seed.)

Benign Races

The high number of real data races for these benchmarks is largely due to benign races on volatile variables used for synchronization — for example, to implement a tournament barrier or a custom lock. Although CalFuzzer does not understand these sophisticated synchronization schemes, our determinism assertions automatically provide some confidence that these races are benign because, over the course of many experiment runs, they did not lead to nondeterministic final results.

Note that it can be quite challenging to verify by hand that these races are benign. On inspecting the benchmark code and these data races, several times we believed we had found a synchronization bug. But on deeper inspection, the code was found to be correct in all such cases.

The number of high-level races is low for the JGF benchmarks because all but `montecarlo` exclusively use volatile variables (and thread joins) for synchronization. Thus, all observable scheduling nondeterminism is due to data races.

The number of high-level races is low for the Parallel Java benchmarks because they primarily use a combination of volatile variables and atomic compare-and-set operations for synchronization. Currently, the only kind of high-level race our modified CalFuzzer recognizes is a lock race. Thus, while we believe there are many (benign) races in the ordering of these compare-and-set operations, CalFuzzer does not report them. The one high-level race for `pi`, indicated in the table and described above, was confirmed by hand.

Discussion

Although our checking of determinism assertions is sound — an assertion failure always indicates that two executions with matching initial states can yield non-matching final states — it is incomplete. Parallelism bugs leading to nondeterminism may still exist even when testing fails to find any determinism violations.

However, in our experiments we successfully distinguished the known harmful races from the benign ones in only a small number of trials. Thus, we believe our determinism assertions can help catch harmful nondeterminism due to parallelism, as well as saving programmer effort in determining whether or not real races and other potential parallelism bugs can lead to incorrect program behavior.

3.5 Discussion

In this section, we compare the concepts of atomicity and determinism. Further, we discuss several other possible uses for bridge predicates and assertions.

Atomicity versus Determinism

A concept complementary to determinism in parallel programs is atomicity. A block of sequential code in a multithreaded program is said to be *atomic* [65] if for every possible interleaved execution of the program there exists an equivalent execution with the same overall behavior in which the atomic block is executed serially (i.e. the execution of the atomic block is not interleaved with actions of other threads). Therefore, if a code block is atomic, the programmer can assume that the execution of the code block by a thread cannot be interfered with by any other thread. This enables programmers to reason about atomic code blocks sequentially. This seemingly similar concept has the following subtle differences from determinism:

1. Atomicity is the property about a sequential block of code — i.e. the block of code for which we assert atomicity has a single thread of execution and does not spawn other threads. Note that a sequential block is by default deterministic if it is not interfered

with by other threads. Determinism is a property of a parallel block of code. In determinism, we assume that the parallel block of code's execution is not influenced by the external world.

2. In atomicity, we say that the execution of a sequential block of code results in the same state no matter how it is scheduled with other external threads, i.e. atomicity ensures that *external nondeterminism* does not interfere with the execution of an atomic block of code. In determinism, we say that the execution of a parallel block of code gives the same semantic state no matter how the threads inside the block are scheduled — i.e. determinism ensures that *internal nondeterminism* does not result in different outputs.

In summary, *atomicity* and *determinism* are orthogonal concepts. Atomicity reasons about a single thread under external nondeterminism, whereas determinism reasons about multiple threads under internal nondeterminism.

Here we focus on atomicity and determinism as program specifications to be checked. There is much work on atomicity as a language mechanism, in which an atomic specification is instead *enforced* by some combination of library, run-time, compiler, or hardware support. One could similarly imagine enforcing determinism specifications through, e.g., compiler and run-time mechanisms [8, 20].

Other Uses of Bridge Predicates

We have already argued that bridge predicates simplify the task of directly and precisely writing deterministic properties in parallel programs. However, we believe that bridge predicates could provide us a simple, but powerful tool to express correctness properties in many other situations. For example, if we have two versions of a program *P1* and *P2* and if we expect them to produce the same output on same input, then we can easily assert this using our framework as follows:

```
deterministic assume(Pre) {
    if (nonDeterministicBoolean()) {
        P1
    } else {
        P2
    }
} assert(Post);
```

where *Pre* requires that the inputs are the same and *Post* specifies that the outputs will be the same.

In particular, if a programmer has written both a sequential and parallel version of a piece of code, he or she can specify that the two versions are semantically equivalent with an assertion like:

```
deterministic assume(A==A' and B==B') {
    if (nonDeterministicBoolean()) {
        C = par_matrix_multiply_int(A, B);
    } else {
        C = seq_matrix_multiply_int(A, B);
    }
} assert(C == C');
```

where `nonDeterministicBoolean()` returns `true` or `false` nondeterministically.

Recall the way we have implemented our determinism checker in Java — we serialize a pair of projections of the input and output states for each execution to the file-system. This particular implementation allows us to quickly write regression tests simply as follows:

```
deterministic assume(Pre) {
    P
} assert(Post);
```

where *Pre* asserts that the inputs are the same and *Post* asserts that the outputs are the same. In the above code, we simply assert that the input-output behavior of `P` remains the same even if `P` changes over time, but maintains the same input-output behavior. The serialized input and output states implicitly store the regression test on the file-system.

Further, we believe there is a wider class of program properties that are easy to write in bridge assertions but would be quite difficult to write otherwise. For example, consider the specification:

```
deterministic assume(set.size() == set'.size()) {
    P
} assert(set.size() == set'.size());
```

This specification requires that sequential or parallel program block `P` transforms `set` so that its final size is the same function of its initial size independent of its elements. The specification is easy to write even in cases where the exact relationship between the initial and final size might be quite complex and difficult to write. It is not entirely clear, however, when such properties would be important or useful to specify/assert.

3.6 Related Work

As we discussed at the beginning of this chapter, there is a large body of work attacking harmful program nondeterminism by detecting data races. There has also been recent work on detecting or eliminating other sources of nondeterminism such as high-level races [122, 9] and atomicity violations [64, 57, 62, 116].

For more than forty years, assertions — formal constraints on program behavior embedded in a program' source — have been used to specify and prove the correct behavior of sequential [66, 78] and parallel [113] programs. More recently, assertions have found widespread use as a tool for checking at run-time for software faults to enable earlier detection and easier debugging of software errors [127, 101]. In this work, we propose bridge assertions, which relate pairs of states from different program executions.

Checking and Verifying Determinism. Siegel, et al., [135] propose a technique for combining symbolic execution with model checking to verify that parallel, message-passing numerical programs compute equivalent answers to their sequential implementations.

Sadowski, et al., [128] propose a different notion of determinism, one that is a generalization of *atomicity*. They say that a parallel computation is deterministic if is both free from external interference (*externally serializable*) and if its threads communicate with each other in a strictly deterministic fashion (*internal conflict freedom*). That is, for a computation to be deterministic not only must it contain no data races, but the partially-ordered sequence of lock operations and other synchronization events must be identical on every execution. These conditions ensure that every schedule produces bit-wise identical results. Further, they propose SingleTrack [128], a sound dynamic determinism analysis that can identify determinism violations in a single execution of a program under test.

This form of determinism used by SingleTrack [128] is much more strict than the determinism proposed in this work. Our determinism specifications can be applied to programs, such as those using locks or shared buffers, in which internal threads communicate nondeterministically, but still produce deterministic final results. Further, we allow users to provide custom predicates to specify what is means for the results of two different thread schedules to be *semantically* deterministic.

DICE [154] is a patent-pending static analysis for verifying determinism. DICE also focuses on a very strict notion of determinism — no task or loop iteration may write to a memory location than any parallel task or loop iteration can read or write. By combining a flow-insensitive pointer analysis with abstract interpretation of array indices (using the polyhedral or octagonal domain) to prove that the memory accesses of each parallel task are independent, DICE can prove the determinism of simplified, structured-parallel versions of several JGF benchmarks.

InstantCheck [110] checks determinism as we have proposed, using hashing to compare final states from different executions. But InstantCheck proposes to perform such hashing incrementally, rather than at the end of the computation, and to use hardware support to accelerate the incremental hashing. InstantCheck's approach is not able to check higher-level semantic equivalence, and requires complex record and replay of allocated/remapped/etc. memory addresses to deal with dynamic memory allocation.

Enforcing Determinism. A number of techniques been proposed for forcing existing multithreaded programs to execute deterministically — i.e., to control program execution so that, on a given input, every run takes an equivalent thread schedule. These techniques,

including Kendo [112], DMP [42], CoreDet [15], Determinator [11], Dthreads [94], Calvin [79], and RCDC [43], employ a variety of compiler, operating system, and hardware approaches to achieve deterministic execution.

This kind of deterministic execution can be very useful in reproducing and debugging parallel errors. Enforced deterministic execution can also make testing more effective, because if a program behaves correctly on a given input during test, then the program will execute identically in production. The particular thread schedule chosen by such systems typically depends on per-thread instruction counts, however, so even small program modifications can change the thread schedule of the entire application, uncovering unrelated and previously-suppressed parallelism errors.

3.7 Summary

We have introduced bridge predicates and bridge assertions for relating pairs of states across different executions. We have shown how these predicates and assertions can be used to easily and directly specify that a parallel computation is deterministic. And we have shown that such specifications can be useful in finding parallel nondeterminism bugs and in distinguishing harmful from benign races. Further, we believe that bridge assertions may have other potential uses.

Chapter 4

DETERMIN: Inferring Likely Determinism Specifications for Multithreaded Programs

In Chapter 3, we argued that programmers should have a way to directly and easily specify that a parallel software application behaves deterministically, despite the nondeterministic interleaving of program threads. We proposed a scheme for asserting that a block of parallel code exhibits the intended, user-specified *semantic determinism*. Formally, our framework allowed a programmer to give a specification for a block P of parallel code as:

```
deterministic assume(Pre(s_0, s'_0)) {
    P
} assert(Post(s, s'));
```

This specification asserts the following: Suppose P is executed twice with potentially different schedules, once from initial state s_0 and once from s'_0 and yielding final states s and s', respectively. Then, if the user-specified *precondition Pre* holds over s_0 and s'_0, then s and s' must satisfy the user-specified *postcondition Post*.

We argued that such assertions allow a programmer to specify the correctness of the use of parallelism in an application independently of the functional correctness. That is, one can specify that different executions of a parallel program on the same input cannot erroneously produce non-equivalent outputs due to scheduling nondeterminism. This can be accomplished without having to specify anything about the correctness of individual outputs in terms of their corresponding inputs. Our experiments showed that if the determinism specification of a parallel program is provided, we can distinguish true races from benign ones in the program and find bugs in parallel programs that arise due to internal nondeterminism.

 A version of this chapter was published in the Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE '10), http://doi.acm.org/10.1145/1806799.1806860

In this chapter, we propose to automatically infer likely determinism specifications for parallel programs. Specifically, given a set of test inputs and thread schedules, for each procedure P of a parallel program, we infer a determinism specification for the body of a procedure P:

```
void P() {
    deterministic assume(Pre(s0, s0')) {
        ... body of P ...
    } assert(Post(s, s'));
}
```

A key challenge in inferring likely determinism specification of a parallel program is that there could be several specifications for the program; however, not all of the specifications are interesting. For example, the following determinism specification holds for any parallel program, where *Pre* is any predicate:

```
void P() {
    deterministic assume(Pre(s0, s0')) {
        ... body of P ...
    } assert(true);
}
```

To address the problem of inferring "interesting" determinism specifications, we argue that a (*Pre*, *Post*) pair is "interesting" if the following two conditions hold.

1. *Pre* is a weakest liberal precondition for *Post* and *Post* is a strongest liberal postcondition for *Pre*, and

2. *Post* is the strongest liberal postcondition for any possible *Pre*, which we show to be unique.

We give an algorithm, DETERMIN, to compute one such "interesting" determinism specification from a set of executions observed on a given set of inputs and schedules. We formally prove that if the set of given inputs and schedules is the set of all inputs and schedules, then we infer an actual "interesting" determinism specification of the program.

We have implemented DETERMIN for Java and applied it to all of our benchmarks from the previous chapter. We were able to infer specifications largely equivalent to or stronger than our existing, manual assertions.

We believe that the inference of determinism specifications can aid in program understanding and documentation of deterministic behavior of parallel programs. Specifically, a correct inferred specification documents for programmers the deterministic aspect of the parallel behavior of an application. Moreover, an unexpected determinism specification can indicate to a programmer the presence of buggy or otherwise unintended behavior. For example, consider a specification indicating a critical component of the program output is *not* deterministic; or consider a specification indicating that a program's determinism hinges on some believed-to-be insignificant portion of the input.

Related Work

There is a rich literature on invariant generation – see, e.g., [50, 63, 7, 75, 163, 95, 165, 6, 144, 166, 98, 142, 69, 40, 70, 162]. Daikon [50] automatically infers likely program invariants using statistical inference from a program's execution traces. Csallner et al. [40] propose an approach, called DySy, that combines symbolic execution with dynamic testing to infer preconditions and postconditions for program methods. Hangal and Lam [75] propose DIDUCE, which uses online analysis to discover simple invariants over the values of program variables. Deryaft [98] is a tool that specializes in generating constraints of complex data structures. Logozzo [95] proposed a static approach that derives invariants for a class as a solution of a set of equations derived from the program source. Houdini [63] is an annotation assistant for ESC/Java [55]. It generates a large number of candidate invariants and repeatedly invokes the ESC/Java checker to remove unprovable annotations, until no more annotations are refuted. The problem of program invariant generation is related to the problem of automatic mining of temporal specifications of programs. Previous work [7, 163, 144, 165, 6, 1, 73, 161] have approached this problem using both dynamic and static analysis techniques. The above mentioned techniques mostly focuses on generation of traditional specifications. Our approach is the first one to infer likely determinism specifications for parallel programs. Unlike traditional specifications, our inferred specifications relate two program states coming from different executions.

Discussion of related work on specifying, checking, and enforcing deterministic behavior can be found in Section 3.6.

4.1 Formal Background

In this section, we review the key features of our previously proposed determinism specifications [29], which are described in detail in Chapter 3. In summary, we previously proposed [29] the following construct for the specification of deterministic behavior:

```
deterministic assume(Pre(s0, s0')) {
    P
} assert(Post(s, s'));
```

This specification states that for any two program states s_0 and s'_0, if

- the determinism precondition $Pre(s_0, s'_0)$ holds
- an execution of P from s_0 terminates and results in state s, and
- an execution of P from s'_0 terminates and results in state s'

then the determinism postcondition $Post(s, s')$ must hold.

More formally, let $P(s_0, \sigma)$ denote the resulting program state if we run procedure P on initial state s_0 and with thread schedule σ. Then, the above determinism specification states that:

$$\forall s_0, s_0', \sigma, \sigma'.\ Pre(s_0, s_0') \Longrightarrow Post(P(s_0, \sigma), P(s_0', \sigma'))$$

We abbreviate this condition by:

$$Pre \implies_P Post$$

Note that, technically, only certain thread schedules are possible for each initial program state. That is, schedules σ should not be universally quantified, but must come from the set $\Sigma(s_0)$ of thread schedules for procedure P realizable from program state s_0. And function $P(s_0, \sigma)$ is defined only for $\sigma \in \Sigma(s_0)$. For simplicity, however, we omit any further references to $\Sigma(s_0)$.

Note also that, for certain initial states s_0 and thread schedules σ, procedure P may not terminate. In this case, function $P(s_0, \sigma)$ is not defined, as there is no resulting program state. We implicitly quantify over only terminating executions. Thus, our determinism specifications are *partial*.

The advantage of our determinism specifications is that they provide a way to specify the correctness of just the use of parallelism in a program, independent of the program's full functional correctness. In many situations, writing a full specification of functional correctness is difficult and time consuming. But, a simple determinism specification enables us to use automated technique to check for parallelism bugs, such as harmful data races causing semantically nondeterministic behavior.

4.2 Overview of DETERMIN

In this section, we give an informal overview of our algorithm for dynamically inferring likely determinism specifications. Consider a procedure, `bestTree`, which, given a collection of DNA sequences, computes in parallel a most likely phylogenetic tree[1]:

```
void bestTree(int N, int[][] dna,
              int &score, int[] &tree)
{
    // Parallel branch-and-bound search (with N threads)
    // for an optimal tree given DNA sequences.
    ...
}
```

Given two runs of this procedure on identical DNA sequence data, we would expect to get identical final likelihood scores. Because the procedure is a parallel branch-and-bound search, we cannot expect two different runs to necessarily compute the same phylogenetic

[1] A tree showing suspected evolutionary relationships — i.e. shared common ancestry — among a group of species or individuals.

tree — for the input data, there may be multiple trees with the same best score. Thus, we might manually specify the deterministic behavior of procedure `bestTree` as:

```
void bestTree(int N, int[][] dna,
              int &score, int[] &tree)
{
    deterministic assume (dna == dna') {
        ...
    } assert (score == score');
}
```

Data Collection

To infer a determinism specification for procedure `bestTree`, we must first collect some sample of representative executions. Suppose the programmers who wrote the code have also constructed two DNA sequence data sets, D_1 and D_2, which they use for testing. (In the absence of hand-constructed test inputs, we could potentially use random or symbolic test generation to construct test cases.) Then, suppose we execute the procedure, perhaps as part of some existing application or test, twice on each input:

$$\begin{aligned}
N = 10,\ dna = D_1 &\longmapsto score = 140,\ tree = t_1 \\
N = 10,\ dna = D_1 &\longmapsto score = 140,\ tree = t_2 \\
N = 10,\ dna = D_2 &\longmapsto score = 175,\ tree = t_3 \\
N = 10,\ dna = D_2 &\longmapsto score = 175,\ tree = t_4
\end{aligned}$$

Specification Inference

In theory, there are infinitely many possible bridge predicates relating pairs of inputs (N, dna), (N', dna') or pairs of outputs $(score, tree)$, $(score', tree')$. But we care only about a very restricted subset of these bridge predicates: predicates that compare only individual components across pairs of inputs or outputs, and that compare those components only for certain types of equality or approximate equality.

For example, for procedure `bestTree`, we are interested only in four bridge predicates as preconditions:

$$true, \quad N = N', \quad dna = dna', \quad N = N' \wedge dna = dna'$$

and four bridge predicates as postconditions:

$$true,\ score = score',\ tree = tree',\ score = score' \wedge tree = tree'$$

More generally, by focusing on equality between components of input and output states, we only have to consider finitely-many possible determinism specifications. (Although the

number of possible specifications is still exponential in the number of input and output variables.)

Thus, we can think of the determinism specification inference problem as having two parts. First, we should determine which of these possible determinism specifications is *consistent* with our observed executions. Second, we must decide which of the consistent specifications to select as the final inferred specification.

There are six possible determinism specifications consistent with the four above observed executions. Four of these specifications are of the form $Pre \Longrightarrow_{\texttt{bT}} true$ — that is, each of the four possible preconditions paired with the trivial postcondition. The other two possible determinism specifications are:

$$(dna = dna') \Longrightarrow_{\texttt{bT}} (score = score') \tag{4.1}$$

$$(N = N' \wedge dna = dna') \Longrightarrow_{\texttt{bT}} (score = score') \tag{4.2}$$

In selecting one of these six potential determinism specifications, we are guided by two principles: (1) First, we should select a specification with as strong of a postcondition as possible. Some parts of a procedures output may be scheduler-dependent and nondeterministic, but we would ideally like a specification that captures all parts of the output that are deterministic. (2) Second, for a given postcondition, we should select as weak of a precondition as possible.

For our running example, two of the possible specifications (i.e. Equations 4.1 and 4.2 above) have the strongest consistent postcondition $score = score'$. (Of course, no consistent postconditions contain $tree = tree'$ because we observed executions with identical inputs but different final values of *tree*.) Selecting the weaker of the two possible consistent preconditions for $score = score'$ gives us the determinism specification:

$$(dna = dna') \Longrightarrow_{\texttt{bT}} (score = score')$$

For this example, the inferred determinism specification is exactly the one we would have manually written. In general, however, there is always the danger that we will infer a postcondition that is too strong because we have observed no executions showing the nondeterminism of some output. Similarly, we may infer a precondition that is too weak because we have observed no executions showing that the deterministic behavior depends on a particular input. In the end, we must rely on having a sufficiently representative set of test inputs and running on sufficiently-many possible thread schedules to defend against inferring inaccurate determinism specifications.

4.3 Inferring Determinism Specifications

In this section, we formally describe the problem of inferring determinism specifications. Let P be a procedure that executes atomically and with internal parallelism. A procedure P in a given program is *atomic* [65] if, no other component of the program that can run in

parallel with P can interfere with the execution of P. We say that procedure P has *internal parallelism* if, when P is executed, P performs a computation in parallel and P returns only after all parallel work has completed. For example, P may spawn several threads, but must join all of the threads before returning.

For the body of a procedure P, we want to infer a determinism specification of the form:

```
void P() {
    deterministic assume(Pre(s0, s0')) {
        ... body of P ...
    } assert(Post(s, s'));
}
```

Determinism Specification Model

In theory, the pre- and postconditions in a determinism specification can be arbitrary bridge predicates. We restrict our attention, however, to a specific class of bridge predicates: conjunctions of *semantic equality* predicates.

We treat programs as having a finite set M of disjoint memory locations $\{m_1, \ldots, m_k\}$. Then, a program state s is a mapping from these global variables m_i to values $s(m_i)$ from set V of possible program values.

We further suppose that we have a finite set EQ of semantic equality predicates on program values. We require these predicates to be reflexive and symmetric relations on program values, and that this set include the strict equality predicate $v = v'$. (In our implementation, for example, we also include a approximate numeric equality $|v - v'| \leq \epsilon$ and semantic object equality $v.\texttt{equals}(v')$.)

Then, we consider the class of bridge predicates characterized by subsets of $M \times EQ$. For some $X \subseteq M \times EQ$, we define a bridge predicate φ_X by:

$$\varphi_X(s, s') \quad = \quad \bigwedge_{(m,eq)\in X} eq\Big(s(m),\, s'(m)\Big)$$

That is, for each pair of memory location m and equality predicate eq we compare the value of m from states s and s' using eq. The bridge predicate is the conjunction of all such equality predicates.

We justify this restriction by noting that this class of bridge predicates sufficed to manually specify the natural deterministic behavior of the benchmarks examined in Chapter 3.

An advantage of this restriction is that there exist only finitely many bridge predicates — one for each of the $2^{|M||EQ|}$ subsets of $M \times EQ$. Thus, there are only $2^{2|M||EQ|}$ possible determinism specifications, consisting of one pre- and one postcondition, for a procedure P.

Specification Inference Problem

As described above, every pair of subsets of $M \times EQ$ defines a possible determinism specification. For a given procedure P, many of these possible specifications may be true. That is, there may be many $pre, post \subseteq M \times EQ$ for which $\varphi_{pre} \Longrightarrow_P \varphi_{post}$.

Here we formally describe which of these true specifications, for a procedure P, we believe is the most natural and interesting choice. In short, we should infer specifications $\varphi_{pre} \Longrightarrow_P \varphi_{post}$ only where φ_{post} is the *strongest liberal postcondition* of φ_{pre} and φ_{pre} is a *weakest liberal precondition* of φ_{post}. Further, of such specifications, we should return one with the unique, strongest possible postcondition φ_{post}. (We will show that such a unique, strongest postcondition must exist.)

Lattice Structure of Determinism Specifications

The subsets of $M \times EQ$ naturally form a complete lattice under the ordering $\subseteq$ and with join $\cup$. This induces a complete lattice on bridge predicates φ_X, with:

$$\varphi_X \sqsubseteq \varphi_Y \iff X \supseteq Y\,, \quad \varphi_X \sqcap \varphi_Y = \varphi_{X \cup Y}$$

Note that the lattice on predicates is reversed — *larger* sets yield *smaller* predicates, and the *meet* of two predicates is the *join* of the corresponding sets. This lattice has least and greatest elements:

$$\begin{aligned} \bot &= \varphi_{M \times EQ} &&= \forall (m, eq) \in M \times EQ.\ eq(s(m), s'(m)) \\ \top &= \varphi_{\emptyset} &&= \mathit{true} \end{aligned}$$

Note that, because every element of EQ is reflexive and symmetric, every predicate φ_X is reflexive and symmetric on program states. In particular, $\bot(s, s)$ for any state s.

We now state several simple but important properties of these lattices and their relation to the validity of determinism specifications.

Proposition 1. *The lattice operations $\sqcap$ and $\sqsubseteq$ on bridge predicates are exactly logical conjunction and implication:*

$$\begin{aligned} \varphi_X \wedge \varphi_Y &\quad = \quad \varphi_X \sqcap \varphi_Y \quad (= \varphi_{X \cup Y}) \\ \varphi_X \implies \varphi_Y &\quad \iff \quad \varphi_X \sqsubseteq \varphi_Y \quad (= X \supseteq Y) \end{aligned}$$

Proposition 2. *Relation $\Longrightarrow_P$ distributes over the meet ($\sqcap$) operation on bridge predicates, and the join operation on subsets of $M \times EQ$, in the sense that:*

$$\varphi_X \Rightarrow_P \varphi_Y \ \wedge \ \varphi_X \Rightarrow_P \varphi_{Y'} \iff \varphi_X \Rightarrow_P (\varphi_Y \sqcap \varphi_{Y'})$$

or, equivalently:

$$\varphi_X \Rightarrow_P \varphi_Y \ \wedge \ \varphi_X \Rightarrow_P \varphi_{Y'} \iff \varphi_X \Rightarrow_P \varphi_{Y \cup Y'}$$

Proposition 3. *Relation $\Longrightarrow_P$ is monotone in its second argument and anti-monotone in its first argument with respect to the lattice on bridge predicates:*

$$\varphi_X \implies \varphi_{X'},\ \varphi_Y \implies \varphi_{Y'} \implies (\varphi_{X'} \Rightarrow_P \varphi_Y \implies \varphi_X \Rightarrow_P \varphi_{Y'})$$

In light of Proposition 3, we will say that a determinism specification (φ_X, φ_Y) is *stronger* or *more strict* than another specification $(\varphi_{X'}, \varphi_{Y'})$ — denoted $(\varphi_X, \varphi_Y) \sqsubseteq (\varphi_{X'}, \varphi_{Y'})$ — when $\varphi_{X'} \implies \varphi_X$ and $\varphi_Y \implies \varphi_{Y'}$.

Strongest Liberal Postcondition

For any precondition φ_{pre} for a procedure P, we can define the *strongest liberal postcondition* $\mathsf{SLP}_P(\varphi_{pre})$ of φ_{pre} as the least φ_{post} such that $\varphi_{pre} \Longrightarrow_P \varphi_{post}$. We show below that there is always a unique $\mathsf{SLP}_P(\varphi_{pre})$.

Proposition 4. *Let φ_{pre} be a precondition for procedure P.*

$$\mathit{SLP}_P(\varphi_{pre}) \quad = \quad \bigsqcap \{\varphi_{post} \mid \varphi_{pre} \Longrightarrow_P \varphi_{post}\}$$

Proof. First, note that $\varphi_{pre} \Longrightarrow_P \top$ so the meet in the proposition is over a non-empty set. Let $\varphi_{\mathtt{slp}}$ denote the meet over all postconditions that follow from φ_{pre}.

Then, $\varphi_{pre} \Longrightarrow_P \varphi_{\mathtt{slp}}$, because $\Longrightarrow_P$ distributes over $\sqcap$.

Further, $\varphi_{\mathtt{slp}}$ is clearly the least φ_{post} such that $\varphi_{\mathrm{pre}} \Longrightarrow_P \varphi_{post}$, because it is the meet of all such postconditions. □

Corollary 1. *Operator SLP_P is monotone.*

That is, if $\varphi_X \implies \varphi_Y$, then $\mathit{SLP}_P(\varphi_X) \implies \mathit{SLP}_P(\varphi_Y)$.

Proof. Suppose $\varphi_X \implies \varphi_Y$.

Because $\varphi_Y \Longrightarrow_P \mathsf{SLP}_P(\varphi_Y)$ and by the anti-monotonicity of $\Longrightarrow_P$, it must be the case that $\varphi_X \Longrightarrow_P \mathsf{SLP}_P(\varphi_Y)$. Therefore, because $\mathsf{SLP}_P(\varphi_X)$ is the strongest postcondition of φ_X, we have $\mathsf{SLP}_P(\varphi_X) \implies \mathsf{SLP}_P(\varphi_Y)$. □

By the monotonicity SLP_P, the strongest postcondition that holds for P under any possible precondition, is $\mathsf{SLP}_P(\bot)$. Note that, equivalently, this unique strongest postcondition is the meet over all true postconditions:

$$\bigsqcap \{\varphi_{post} \mid \exists \varphi_{pre}.\ \varphi_{pre} \Longrightarrow_P \varphi_{post}\}$$

Thus, in particular, postcondition $\mathsf{SLP}_P(\bot)$ is the conjunction of the most individual equality predicates of any true postcondition.

Weakest Liberal Precondition

We can similarly define the *weakest liberal precondition* of a postcondition φ_{post}. However, because we restrict our preconditions and postconditions to be conjunctions of equality predicates on individual memory locations, there may not be a unique weakest (or largest) precondition for a φ_{post}. Thus, we must define $\texttt{WLP}_P(\varphi_{post})$ to be the set of all weakest liberal preconditions:

Definition 2. *$\varphi_{pre} \in \texttt{WLP}_P(\varphi_{post})$ — i.e. is a weakest liberal precondition of φ_{post} — if and only if both:*

1. *$\varphi_{pre} \Longrightarrow_P \varphi_{post}$, and*
2. *If there exists a φ' such that $\varphi' \Longrightarrow_P \varphi_{post}$ and $\varphi_{pre} \implies \varphi'$, then $\varphi' = \varphi_{pre}$.*

Inferred Determinism Specification

With these formal definitions, we can say that the determinism specification inference problem for a procedure P is to compute, or to approximate as closely as possible, a determinism specification $\varphi_{pre} \Longrightarrow_P \varphi_{post}$ where $\varphi_{post} = \texttt{SLP}_P(\bot)$ is the unique strongest possible postcondition for any precondition and where φ_{pre} is a weakest liberal precondition of φ_{post}.

4.4 DETERMIN Algorithm

In the previous section, we have defined the set of strongest true determinism specifications for a given procedure P. When inferring a determinism specification from a limited number of executions of a procedure P, however, we can only approximate the procedure's true specification.

Suppose we have a finite set R of observed executions $\{(s_1, \sigma_1, t_1), \ldots, (s_n, \sigma_n, t_n)\}$ of procedure P, where each t_i is the state $P(s_i, \sigma_i)$ resulting from executing P from initial state s_i on thread schedule σ_i. A determinism specification $(Pre, Post)$ is satisfied *for the observed executions* R, which we abbreviate $Pre \Longrightarrow_{P,R} Post$, when:

$$\forall_{1 \le i,j \le n}.\ Pre(s_i, s_j) \implies Post(t_i, t_j)$$

Note that this definition is identical to that of $\Longrightarrow_P$, except that we only universally quantify over the observed inputs and thread schedules, rather than quantifying over all possible inputs and schedules. We can similarly define the strongest liberal postcondition $SLP_{P,R}$ and weakest liberal preconditions $WLP_{P,R}$ over observed executions R.

Our overall inference algorithm is presented in Algorithm 1. Given a set of executions R of a procedure P, we will infer a likely determinism specification $(Pre_R, Post_R)$.

The algorithm consists of two stages. First, we infer $Post_R$ by computing $\texttt{SLP}_{P,R}(\bot)$, the strongest liberal postcondition, given executions R of P, of precondition $\bot$. Recall that this

is the strongest possible postcondition, given executions R, for any precondition. Second, we infer Pre_R by computing $\texttt{WLP}_{P,R}(Post_R)$, a weakest liberal precondition, given executions R of P, of postcondition $Post_R$.

Computing the Strongest Postcondition

Algorithm 2 computes the strongest liberal postcondition, given executions R, of some φ_{pre}. The algorithm iterates over every pair of executions $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j)$ that satisfy φ_{pre}. For each such pair, it computes the set of all individual equality predicates that hold on the resulting program states. The algorithm accumulates into *post* the intersection of all these sets. Thus, at the end of the algorithm, φ_{post} is the conjunction of all equality predicates that hold for pairs of post-states resulting from pre-states matching φ_{pre}. That is, φ_{post} is the strongest liberal postcondition of φ_{pre} for the observed executions R.

Checking the condition at Line 3 and computing the set and the intersection in Line 4 can all be done in $O(|M||EQ|)$ time. Thus, as these steps must be performed once for each pair of executions, the whole SLP computation requires $O(|M||EQ||R|^2)$ time.

Algorithm 1 Infer a likely determinism specification for a procedure P, given a set R of executions of procedure P.

1: $Post_R \leftarrow \texttt{SLP}_{P,R}(\bot)$
2: $Pre_R \leftarrow \texttt{WLP}_{P,R}(Post_R)$
3: **return** $(Pre_R, Post_R)$

Algorithm 2 Compute the strongest liberal postcondition $\texttt{SLP}_{P,R}(\varphi_{pre})$ of φ_{pre}.

1: $post \leftarrow M \times EQ$
2: **for all** $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j) \in R \times R$ **do**
3: **if** $\varphi_{pre}(s_i, s_j)$ **then**
4: $post \leftarrow post \cap \{(m, eq) \mid eq(t_i(m), t_j(m))\}$
5: **end if**
6: **end for**
7: **return** φ_{post}

Algorithm 3 Compute a weakest liberal precondition $\texttt{WLP}_{P,R}(\varphi_{post})$ of φ_{post}.

Require: $\bot \Longrightarrow_{P,R} \varphi_{post}$
1: $pre \leftarrow M \times EQ$
2: **for all** $(m, eq) \in M \times EQ$ **do**
3: **if** $\varphi_{pre-\{(m,eq)\}} \Longrightarrow_{P,R} \varphi_{post}$ **then**
4: $pre \leftarrow pre - \{(m, eq)\}$
5: **end if**
6: **end for**
7: **return** φ_{pre}

Computing a Weakest Precondition

Algorithm 3 computes a weakest liberal precondition, given executions R, for some φ_{post}. The algorithm begins with $\varphi_{pre} = \bot = \varphi_{M \times EQ}$, and then greedily weakens φ_{pre} until it can be made no weaker while remaining a precondition for φ_{post} on the observed executions R. Lines 3-5 check if the current φ_{pre} can be safely weakened by removing the conjunct $eq(s(m), s'(m))$ from $\varphi_{pre}(s, s')$.

It is sufficient to consider each (m, eq) only once during the computation. Suppose it was not possible to weaken some pre_1 by removing (m, eq), but it was possible to weaken a later pre_2 by removing the same (m, eq). Because pre_2 comes later, $pre_1 \supseteq pre_2$ and thus $(pre_1 - \{(m, eq)\}) \supseteq (pre_2 - \{(m, eq)\})$ But, then if $\varphi_{pre_2 - \{(m,eq)\}} \Longrightarrow_{P,R} \varphi_{post}$, we must also have $\varphi_{pre_1 - \{(m,eq)\}} \Longrightarrow_{P,R} \varphi_{post}$, which is a contradiction.

Note that, depending on the order in which the algorithm considers the elements of $M \times EQ$, it can return any of the possible weakest preconditions of φ_{post} under the observed executions.

Checking the condition at Line 3 requires $O(|M||EQ||R|^2)$ time, to determine on every pair of observed executions that $\varphi_{pre - \{(m,eq)\}} \implies \varphi_{post}$. Thus, the entire computation of a Pre_R requires $O(|M|^2|EQ|^2|R|^2)$ time.

Correctness

We now formally state several important properties of our determinism specification inference algorithm.

Most importantly, we show in Proposition 5 that the DETERMIN algorithm is correct. That is, for any inferred determinism specification $(Pre_R, Post_R)$ for executions R of procedure P:

1. Pre_R is a weakest liberal precondition for $Post_R$ and $Post_R$ is a strongest liberal postcondition for Pre_R, given the executions in R.

2. $Post_R$ is the unique strongest liberal postcondition for any possible precondition given the executions in R.

We further show (Corollary 4) that an inferred postcondition $Post_R$ will always be stronger than the strongest true postcondition $\mathtt{SLP}_P(\bot)$. And the more executions R we observe, the weaker — i.e. closer to the true strongest postcondition — our inferred postcondition will be (Proposition 6).

Example 5 shows that we cannot make analogous guarantee for our inferred precondition Pre_R. Rather, we can only guarantee that additional executions will only strengthen the inferred precondition *as long as they do not weaken the postcondition* $Post_R$ (Propositions 7 and 8). And, if $Post_R$ is the true strongest postcondition for any precondition and for all executions, then as we observe additional executions our stronger and stronger inferred Pre_R will approach a true weakest precondition for $Post_R$ (Corollaries 6 and 7).

In the results below, we make liberal use of the fact that all of properties shown in Section 4.3 for $\Longrightarrow_P$, $\mathtt{SLP}_P$, and $\mathtt{WLP}_P$ all hold $\Longrightarrow_{P,R}$, $\mathtt{SLP}_{P,R}$, and $\mathtt{WLP}_{P,R}$, respectively. (We simply replace universal quantification in the proofs of the respective properties with quantification over the set R.)

Proposition 5. *Let $(Pre_R, Post_R)$ be the specification inferred for executions R of P. Then, $Pre_R \in \mathtt{WLP}_{P,R}(Post_R)$ and $Post_R = \mathtt{SLP}_{P,R}(Pre_R)$.*

Further, for any $\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$, we have $Post_R \implies \varphi_{post}$.

Proof. We have already shown in Section 4.4 that DETERMIN returns a Pre_R that is a weakest liberal precondition of $Post_R$ under executions R.

We shown in Section 4.4 that DETERMIN computes a $Post_R$ equal to $\mathtt{SLP}_{P,R}(\bot)$. Because $\bot \implies Pre_R$ and $\mathtt{SLP}_{P,R}$ is monotone, $Post_R \implies \mathtt{SLP}_{P,R}(Pre_R)$. Therefore, it must be the case that $Post_R = \mathtt{SLP}_{P,R}(Pre_R)$.

Further, suppose $\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$ for some φ_{pre} and φ_{post}. Then, because $\bot \implies \varphi_{pre}$, by the monotonicity of $\mathtt{SLP}_{P,R}$ we have $Post_R \implies \varphi_{post}$. □

Lemma 3. *Let φ_X and φ_Y be bridge predicates such that $\varphi_X \Longrightarrow_{P,R'} \varphi_Y$. Then, $\varphi_X \Longrightarrow_{P,R} \varphi_Y$ for any $R \subseteq R'$.*

Proof. Suppose $\varphi_X \Longrightarrow_{P,R'} \varphi_Y$ and $R \subseteq R'$.

Then, $\varphi_X(s_i, s_j) \implies \varphi_Y(t_i, t_j)$ for all pairs of executions $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j) \in R' \times R'$. Thus, we also have $\varphi_X(s_i, s_j) \implies \varphi_Y(t_i, t_j)$ for any pair of executions from $R \times R$, as $R \times R \subseteq R' \times R'$.

Therefore, $\varphi_X \Longrightarrow_{P,R} \varphi_Y$. □

Proposition 6. *Let $Post_R$ and $Post_{R'}$ be the inferred postconditions for $R \subseteq R'$. Then, $Post_R \implies Post_{R'}$.*

Proof. Note that:

$$\begin{aligned} Post_R &= \bigsqcap \{\varphi_{post} \mid \bot \Longrightarrow_{P,R} \varphi_{post}\} \\ Post_{R'} &= \bigsqcap \{\varphi_{post} \mid \bot \Longrightarrow_{P,R'} \varphi_{post}\} \end{aligned}$$

Because $R \subseteq R'$, by Lemma 3, for every φ_{post} for which $\bot \Longrightarrow_{P,R'} \varphi_{post}$, we also have $\bot \Longrightarrow_{P,R} \varphi_{post}$.

Therefore, $Post_R \implies Post_{R'}$, because $Post_R$ is the meet over an equal or larger set of bridge predicates. □

Corollary 4. *Let $Post_R$ be the inferred postconditions for observed executions R. Then, $Post_R \implies \mathtt{SLP}_P(\bot)$. That is, $Post_R$ is stronger than the strongest true postcondition.*

Example 5. *Consider the following contrived procedure operating on two global variables x and y:*

```
example() {
    <x = x + 1> || <y = 0> || <y = y + 1>;
}
```

Procedure `example` *runs three atomic statements in parallel: an increment of x, an assignment of y to zero, and an increment of y. Suppose we observe the executions:*

$$\begin{aligned} x = 0,\ \ y = 0 \ &\longmapsto\ x = 1,\ \ y = 0 \\ x = 0,\ \ y = 1 \ &\longmapsto\ x = 1,\ \ y = 1 \\ x = 1,\ \ y = 1 \ &\longmapsto\ x = 2,\ \ y = 0 \end{aligned}$$

Then, we will infer the specification precondition $x = x' \wedge y = y'$ and postcondition $x = x' \wedge y = y'$.

But, suppose we observe the additional execution:

$$x = 0,\ \ y = 0 \ \longmapsto\ x = 1,\ \ y = 1$$

Then we will see that $y = y'$ cannot be guaranteed, and we will infer the true specification $x = x' \Longrightarrow_{\texttt{example}} x = x'$, which has a weaker precondition.

Proposition 7. *Let Post be the inferred postcondition for both R and R', with $R \subseteq R'$. Further, let Pre_R be an inferred precondition under R. Then, there is no strictly weaker inferred precondition Pre'_R.*

Proof. Suppose there were some inferred precondition $Pre_{R'} \Longrightarrow_{P,R'} Post$ with $Pre_R \implies Pre_{R'}$. By Lemma 3, we have $Pre_{R'} \Longrightarrow_{P,R} Post$. But Pre_R is a weakest precondition for $Post$ under observed executions R, so $Pre_{R'} = Pre_R$. □

Proposition 8. *Let Post be the inferred postcondition for both R and R', with $R \subseteq R'$. Further, let $Pre_{R'}$ be an inferred precondition under R'. Then, there is a Pre_R from $\mathtt{WLP}_{P,R}(Post)$ — i.e. a possible inferred precondition for observed executions R — such that $Pre_{R'} \implies Pre_R$.*

Proof. By Lemma 3, we have that $Pre_{R'} \Longrightarrow_{P,R} Post$. Thus, either $Pre_{R'}$ is itself a weakest liberal precondition for $Post$ under R, or else there is some $Pre_R \in \mathtt{WLP}_{P,R}(Post)$ such that $Pre_{R'} \implies Pre_R$. □

Corollary 6. *Let the postcondition inferred for executions R be $Post = \mathtt{SLP}_P(\bot)$. Further, let Pre_R be an inferred precondition under R. Then, there are no true preconditions of Post, i.e. elements of $\mathtt{WLP}_P(Post)$, strictly weaker than Pre_R.*

Corollary 7. *Let the postcondition inferred for executions R be $Post = \mathtt{SLP}_P(\bot)$. Further, let Pre be a true precondition for Post. Then, there is a Pre_R from $\mathtt{WLP}_{P,R}(Post)$ — i.e. a possible inferred precondition under observed executions R — such that $Pre \implies Pre_R$.*

A More Conservative Precondition

Our algorithm for computing a precondition from $\texttt{WLP}_{P,R}(Post_R)$ finds a weakest liberal precondition Pre_R such that no pair of executions from R falsifies $Pre_R \Longrightarrow_P Post_R$. When only a small number of executions or procedure inputs are examined, such a precondition may be too weak.

For example, consider a procedure P whose input consists of ten integers $x_0, \ldots, x_9$ and whose output is the sum *sum* of the integers. Suppose we observe executions R of this method from only two distinct initial states — one where $x_0 = \cdots = x_9 = 0$ and one where $x_0 = \cdots = x_9 = 1$. Then, the determinism specification $x_3 = x_3' \Longrightarrow_{P,R} sum = sum'$ is consistent with the data. That is, we observe no pair of executions that falsifies that $x_3 = x_3'$ is a necessary precondition for determinism — i.e. a pair in which $x_3 = x_3'$, but because some other input is not equal, the final *sum*s are not equal.

To combat such an inadequate test set, rather than report any weakest liberal precondition consistent with out observed executions, we can report a *weakest* ***occurring*** *liberal precondition.* We say that a precondition *occurs* (or is *occurring*) if:

Definition 8. *Precondition φ_{pre}* ***occurs*** *in a set R of observed executions iff there is a pair $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j)$ from R, with $i \neq j$, such that φ_{pre} is the strongest bridge predicate satisfied by s_i and s_j. That is, pre is the set $\{(m, eq) \mid eq(s_i(m), s_j(m))\}$.*

Algorithm 4 Compute a weakest liberal occurring precondition $\texttt{WLOP}_{P,R}(\varphi_{post})$ of determinism precondition φ_{post}.

Require: $\bot \Longrightarrow_{P,R} \varphi_{post}$

// Find the occurring preconditions of φ_{post}.
$occurs \leftarrow \emptyset$
for all $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j) \in R \times R$ **do**
  $pre \leftarrow \{(m, eq) \mid eq(s_i(m), s_j(m))\}$
  if $\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$ **then**
    $occurs \leftarrow occurs \cup \{pre\}$
  end if
end for
// Select a weakest occurring precondition of φ_{post}.
for all $pre \in occurs$ **do**
  if $\neg\exists pre' \in occurs.\ pre' \subseteq pre$ **then**
    return φ_{pre}
  end if
end for

We define the set $\texttt{WLOP}_{P,R}(\varphi_{post})$ of weakest liberal occurring preconditions for P of φ_{post} under observed executions R by:

Definition 9. *$\varphi_{pre} \in \texttt{WLOP}_{P,R}(\varphi_{post})$ iff:*

1. *$\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$,*
2. *φ_{pre} occurs in R, and*
3. *If φ'_{pre} occurs in R and $\varphi'_{pre} \Longrightarrow_{P,R} \varphi_{post}$ and $\varphi_{pre} \implies \varphi'_{pre}$, then $\varphi'_{pre} = \varphi_{pre}$.*

Algorithm 4 computes an element of $\texttt{WLOP}_{P,R}$ for a postcondition. We can compute an occurring weakest precondition Pre_R by applying Algorithm 4 to $Post_R$.

Note that, unlike with a `WLP`, observing additional executions may strengthen or weaken $\texttt{WLOP}_{P,R}(Post_R)$, even if $Post_R$ does not change. This is because additional observations can now provide a weaker occurring precondition, in addition to falsifying a previous weakest precondition. However, in the limit of observing all possible executions of P, there is clearly no difference between $\texttt{WLOP}_{P,R}(Post_R)$ and $\texttt{WLP}_{P,R}(Post_R)$.

4.5 Experimental Evaluation

In this section, we describe our efforts to experimentally evaluate the effectiveness of our algorithm for inferring likely determinism specifications. We aim to show that, given a small number of representative executions, our algorithm can infer correct and useful determinism specifications. That is, that our inferred specifications capture the intended natural deterministic behavior of parallel programs.

To evaluate these claims, we implemented our specification inference algorithm DETERMIN for Java applications and applied DETERMIN to the benchmarks to which we previously had manually added determinism specifications in Chapter 3. We then compared the quality and accuracy of the inferred and manual specifications.

Benchmarks

We evaluate DETERMIN on the benchmarks previously examined in Chapter 3. These benchmarks are primarily from the Java Grande Forum (JGF) benchmark suite [46] and the Parallel Java (PJ) library [86]. The names and sizes of the benchmarks are given in Table 4.1. Benchmark `tsp` is a parallel Traveling Salesman branch-and-bound search [122]. The JGF benchmarks include five parallel computation kernels — for successive order-relaxation (`sor`), sparse matrix-vector multiplication (`sparsematmult`), computing the coefficients of a Fourier series (`series`), encryption and decryption (`crypt`), and LU factorization (`lufact`) — as well as a parallel molecular dynamic simulator (`moldyn`), ray tracer (`raytracer`), and Monte Carlo stock price simulator (`montecarlo`). The Parallel Java (PJ) benchmarks include an

app for computing a Monte Carlo approximation of π (`pi3`), an app for cryptographic cracking a cryptographic key (`keysearch3`), an app for parallel rendering of a Mandelbrot Set image (`mandelbrot`), and a parallel branch-and-bound search for an optimal phylogenetic tree (`phylogeny`). These benchmarks range from a few hundred to a few thousand lines of code, with the PJ benchmarks relying on an additional roughly 15,000 lines of library code from the Parallel Java Library for threading, synchronization, and other functionality.

In Chapter 3, we added a single determinism specification block to each benchmark, around the benchmark's entire parallel computation.

Methodology

In order to apply the DETERMIN algorithm to these benchmarks, we need: (1) to decide for which regions in each benchmark to infer determinism specifications, (2) to select a set of representative executions R of these regions as inputs to DETERMIN, (3) to define the sets of *memory locations* M and *semantic equality predicates* EQ for the benchmarks.

Regions for Determinism Specification Inference

In this chapter, we have proposed inferring determinism specifications for procedures — either for all procedures detected to have internal parallelism or for some set of user specified procedures. Our manual determinism specifications in Chapter 3, however, were written not at procedure boundaries, but around certain hand-chosen syntactic blocks of code containing internal parallelism. (Each such block is atomic because it is the only region in its benchmark that performs a parallel computation.) Thus, to enable a fair and direct comparison, we use DETERMIN to infer determinism preconditions and postconditions at the beginning and end of the single deterministic block manually identified in Chapter 3. That is, in each representative execution we record the program state at the beginning and end of the manually identified deterministic block.

Representative Executions

We similarly ran each PJ benchmark and `tsp` twenty times — ten on each of of two selected inputs, half with five threads and half with ten threads. Benchmark `tsp`, all of the JGF benchmarks, and many of the PJ benchmarks come with test inputs. When available, we used two of these test inputs. Otherwise, we constructed inputs by hand. The representative executions were run under the Sun JDK 6 on an eight-core Intel Xeon 2GHz Linux system.

Note that, due to the small number of test inputs, we compute the more conservative weakest liberal *occurring* precondition (WLOP), described in Algorithm 4, for our inferred postcondition, rather than a weakest liberal precondition (WLP).

Memory Locations and Equality Predicates

For the Java program states recorded during the representative executions, we generate a set M of memory locations by enumerating all paths of field dereferences, up to some fixed length, through the programs' memory graphs starting at the local variables and static classes. (For example, n, *this.results.bestScore*, or *AppClass.N_THREADS*.) For completeness, we considered all paths of length up to 8, yielding from roughly 20 to 150 memory locations for each benchmark.

We use several equality predicates to compare these memory locations: Primitive types are compared using strict equality or approximate equality (equal to within 10^{-10}) for floating-point values. Objects are compared using their *equals*() methods. Object arrays, Lists, and Iterables can be compared element-by-element or compared as sets of elements.

Implementation

To capture and record program states at desired points in our benchmarks, the data collection component of our implementation uses the Java Reflection API to traverse and serialize a running program's memory graph. We manually instrumented the local variables in scope at the open and close of each deterministic block.

The specification inference portion of our implementation takes a set of these serialized and pre- and post-states as input and outputs an inferred strongest liberal postcondition and weakest liberal occurring precondition for determinism. Both components together are implemented in roughly 1000 lines of Java code.

Heuristics

The above approach generates a large number of memory locations and equality predicates, leading to determinism specifications with too many conjuncts in their preconditions and postconditions. We employ several heuristics to decrease the size and increase the relevancy of our determinism specifications:

First, we remove from the inferred postconditions any locations not modified in at least one execution by the region of code under examination. Without this heuristic, the strongest postcondition (and thus also the precondition) for a region will contain a conjunct $v = v'$ for each variable v not modified by the region. While such an added conjunct is correct — we can guarantee the determinism of variables that are not modified — it is generally not relevant to computation being performed. On each of our benchmarks, this heuristic removes roughly from 10 to 60 conjuncts.

Second, we remove from the inferred precondition and postcondition any conjuncts that are satisfied by every pair of observed program executions. These locations tend to be global constants, such as hard-coded parameters and Class objects. As above, predicates involving such constants are typically not relevant. On each our benchmarks, this heuristic can remove as many as 75 conjuncts from the precondition or postcondition.

Third, we eliminate redundant conjuncts. For example, if a precondition contains the conjunct $o.equals(o)$ for an array o, then we will not include the redundant, weaker conjunct $o.f.equals(o.f')$. Or if our postcondition contains conjunct $x = x'$, we will not add the redundant conjunct $|x - x'| \leq 10^{-10}$. On each our benchmarks, this heuristic removes only a handful of conjuncts from the final preconditions and postconditions.

Results

The results of our experimental evaluation are shown in Table 4.1. We will argue the these results provide evidence for our claims that DETERMIN can automatically infer determinism specifications that are both accurate and useful.

Accuracy: Postconditions

For every benchmark but `lufact`, our automatically inferred postcondition was at least as strong as the corresponding manually-specified postcondition from Chapter 3. Further, the inferred postcondition for `lufact` is actually more accurate than our manual one. When writing the manual specification for `lufact` in Chapter 3, we wrote postcondition $a = a' \wedge ipvt = ipvt' \wedge x = x'$. But, in fact, the `lufact` routine writes no output into variable x. The relevant output — the solution to the linear system being solved — is written to variable b. The correct postcondition, inferred by DETERMIN, is $a = a' \wedge ipvt = ipvt' \wedge b = b'$

Of the other benchmarks, for all but three of them (`sor`, `moldyn`, and `tsp`), the inferred postcondition is equivalent to the manual one. Although the inferred postconditions contain more conjuncts, these postconditions hold for the same pairs of executions. For example, the manual postcondition for `mandelbrot` is simply $matrix = matrix'$. That is, the resulting image, stored as a matrix of hues, is deterministic. The inferred postcondition also contains $image.myWidth = image.myWidth'$. But this field always holds the width of $matrix$, and thus this conjunct does not strictly strengthen the postcondition.

Further, for benchmarks `sor` and `moldyn`, the inferred postconditions are still correct and are only slightly stronger than the previous manual ones. Both benchmarks retain various intermediate results past the end of their computations. Roughly speaking, our manual assertions for these benchmarks specify that the final answer is independent of the number of threads used, while the inferred specifications capture that these intermediate results are also deterministic for any fixed number of threads.

Accuracy: Preconditions

For all but two benchmarks (`sor` and `sparsematmult`), our inferred preconditions are also as strong as our previous manual determinism specifications. Further, these inferred preconditions, except for `moldyn`'s and `keysearch3`'s, are equivalent to the manual ones although they contain more conjuncts.

Benchmark		Approximate Lines of Code (App + Library)	Precondition			Postcondition		
			# Manual Conjuncts	# Inferred Conjuncts	As Strong As Manual?	# Manual Conjuncts	# Inferred Conjuncts	As Strong As Manual?
JGF	`sor`	300	3	2	No	1	7	Yes
	`sparsematmult`	700	4	4	No	1	2	Yes
	`series`	800	1	3	Yes	1	1	Yes
	`crypt`	1100	1	5	Yes	2	2	Yes
	`moldyn`	1300	2	14	Yes	3	7	Yes
	`lufact`	1500	4	9	Yes	3	3	No*
	`raytracer`	1900	2	3	Yes	1	1	Yes
	`montecarlo`	3600	1	2	Yes	1	1	Yes
PJ	`pi3`	150 + 15,000	2	3	Yes	1	1	Yes
	`keysearch3`	200 + 15,000	3	5	Yes	1	3	Yes
	`mandelbrot`	250 + 15,000	7	11	Yes	1	5	Yes
	`phylogeny`	4400 + 15,000	3	5	Yes	2	11	Yes
`tsp`		700	1	3	Yes	1	2	Yes

Table 4.1: Results of our experimental evaluation of DETERMIN. For each benchmark, we report the approximate size of the benchmark and the number of conjunctions in the manual determinism precondition and postcondition added to the benchmark in Chapter 3. We also report the number of conjuncts in the strongest liberal postcondition (SLP) and weakest liberal occurring precondition (WLOP) of the determinism specification inferred by DETERMIN for each benchmark. Further, we indicate whether each inferred precondition and postcondition is at least as strict as its corresponding hand-specified condition.

The inferred precondition for `moldyn` contains $nthreads = nthreads'$, making it stronger than in our manual specification.

The stronger precondition for `keysearch3` actually highlights an error in the manual specification from Chapter 3. One of the inputs (*partialkey*) to the main computation is missing from the manual precondition. But the conjunct $partialkey = partialkey'$ correctly appear in the inferred precondition.

Limitations

For the `sor` benchmark, our inferred precondition is missing two input parameters on which the deterministic behavior depends. DETERMIN fails to include these two parameters because they each take on the same value in all of JGF test inputs for `sor`. Thus, DETERMIN sees no evidence that these parameters are important for determinism and removes them via our second heuristic. This example shows the need for a sufficiently diverse set of test inputs and executions in order to infer accurate determinism specifications.

Similarly, the postcondition for `tsp` is incorrectly too strong, requiring that two runs on the same input return the same tour. In fact, two such runs could return different tours with the same minimal cost, but our particular test inputs appear to have unique solutions.

Discussion

For nearly all of our benchmarks, DETERMIN infers determinism preconditions and postconditions equivalent to, slightly stronger than, or more accurate than those in our previous, manual specifications. Thus, we argue that DETERMIN can capture the intended and natural deterministic behavior of parallel programs.

Further, although our automatically inferred specifications are somewhat larger than the manual ones from Chapter 3, the total number of inferred conjuncts remains quite small. In particular, we believe that pre- and postconditions with 5 to 15 conjuncts are small enough to be fairly easily understood by a programmer. Thus, we argue that such inferred specifications can help document the deterministic behavior of a routine or application for a programmer. For example, the inferred specification for `lufact` corrected our misunderstanding of the benchmark's behavior.

Further, we argue that such automatically-inferred determinism specifications can be useful in discovering parallelism bugs through anomaly detection. That is, from observing "normal" program executions, DETERMIN infers a specification of the typical, expected deterministic behavior of a program. Then, if more in-depth testing finds executions that are anomalous — i.e. that violate the inferred specification — then those executions may exhibit bugs.

In Chapter 3, we combined determinism specifications with a parallel software testing tool in order to distinguish benign from harmful races in these benchmarks. The specifications inferred by DETERMIN in this chapter are sufficiently similar to those manual specifications to serve the same purpose. In particular, these specifications would allow us to distinguish the harmful data race that exists in the `raytracer` benchmark from the other benign races.

4.6 Summary

We have proposed DETERMIN, an algorithm to dynamically infer a likely determinism specification for a parallel program given a set of example executions. We have implemented DETERMIN in a tool for Java and applied it to several parallel Java benchmarks. The determinism specifications automatically inferred by DETERMIN for these benchmarks were found to mostly be equivalent to or more accurate than determinism specifications written manually in previous work. Thus, we believe that inferred determinism specifications can aid in program understanding, as well as bug detection, for parallel programs.

Chapter 5

Specifying and Checking Semantic Atomicity for Multithreaded Programs

In Chapter 3, we used our *bridge predicates* [29] to specify and to test the natural and intended deterministic behavior of parallel software. But, as we discussed in Chapter 2, some parallel applications can intentionally have nondeterministic behavior, correctly returning different results for different thread schedules. We cannot specify the correctness of such nondeterministic parallelism with our determinism specs. Further, determinism specifications are applicable only to closed programs — i.e., those that use parallelism *internally* to perform some specific computation, producing an output given an input. We cannot write determinism specifications for open programs or libraries, such as, e.g., a concurrent list data structure. Thus, in this chapter, we propose *semantic atomicity* specifications — lightweight specification for parallelism correctness that use bridge predicates to generalize *atomicity*.

Atomicity [64] is a fundamental parallel correctness property for multithreaded programs. A block of code is *atomic* if it appears to execute all at once, indivisibly and without interruption from any other program thread. The behavior of an *atomic* code block can be understood and reasoned about *sequentially*, as no parallel operations can interfere with its execution.

Many researchers have proposed using *transactional memory* hardware, libraries, and/or language constructs to implement such atomic blocks. But in much existing multithreaded code, desired atomicity is implemented using a variety of synchronization techniques, including coarse or fine-grained locking and non-blocking synchronization with primitives such as atomic compare-and-swap. Correctly implementing atomicity using these techniques can be difficult and error-prone. Thus, as we discuss in Section 5.5, there has been great interest in techniques enabling programmers to specify what fragments of their concurrent programs

behave as if atomic, and in techniques for testing or verifying that such programs conform to their atomicity specifications.

Traditional notions of atomicity are often too strict in practice, however, because they require the existence of serial executions that result in an state identical to that of the interleaved execution. We propose an assertion framework [28] that allows programmers to specify that their code is *semantically* atomic — that any parallel, interleaved execution of an atomic block will have an effect *semantically equivalent* to that of executing the block serially. Programmers specify this semantic equivalence using *bridge predicates* [29] (described in Chapter 3) — predicates relating pairs of program states from the interleaved and the equivalent serial execution. Such predicates allow the equivalence of executions to be defined at the level of abstraction of an application.

We further propose an approach [28] to check our semantic atomicity specifications by testing whether or not specified programs are *semantically linearizable*. We choose to check linearizability because (1) this stronger notion is significantly easier to check since the restriction on allowed serial executions significantly reduces the space of serial executions that we must search, and (2) the notion of linearizability is often used to describe the parallel correctness of various concurrent data structures. Essentially, to test linearizability for a particular interleaving we need to consider only permutations of atomic blocks that overlapped in the interleaved execution.

The key to the efficiency of our approach is based on two observations. First, linearizability can be checked efficiently for a parallel execution in which only a small number of atomic blocks overlap, since we need to examine only a small number of similar sequential executions. Second, our experience shows that most atomicity bugs can be reproduced with a small number of overlapping atomic blocks. Thus, we test linearizability of a program by generating parallel executions with only a small number of interrupted atomic blocks. Our experiments show that we can effectively find serious atomicity errors in our benchmarks by testing such *interruption-bounded* executions.

To further reduce the search space for the linearized execution, we propose a set of sound heuristics and optional user annotations. We show in our experiments that in the presence of such annotations we can often find the linearized execution in the first attempt.

We have implemented our assertion framework for Java and used it to specify the intended atomicity of a number of benchmarks. We found that the ability to specify atomicity at the *semantic* level, using bridge predicates, is crucial for capturing the intended atomic behavior of many of our of the benchmarks. Such benchmarks contain sections of code that, while semantically atomic, are not atomic under more strict, traditional notions of atomicity.

In summary, we describe the following contributions:

- We propose using *bridge predicates* to specify that regions of parallel programs are intended to be *semantically atomic*. Our notion of *semantic atomicity* is more general than traditional strict notions of atomicity and is applicable to a wider range of parallel programs.

- We propose an approach to test efficiently and effectively a program's semantic atomicity specification by checking the *linearizability* of program executions with a bounded number of interrupted atomic sections. We further propose program annotations and corresponding heuristics to reduce significantly the search space that must be explored during testing, without sacrificing any ability to find atomicity errors.
- We implement an assertion framework for Java for specifying and testing semantic atomicity specifications and evaluate our approach on a number of Java benchmarks. We find that bridge predicates are required in a majority of the examples. We show that the heuristics we propose make the testing approach both reasonably efficient and effective at finding bugs.
- We find a number of previously unknown atomicity errors, including several in Java's built-in data structure libraries.

5.1 Specifying Semantic Atomicity

In this section, we informally describe atomicity and motivate our proposal for semantic atomicity specifications. We first describe a real-world motivating example. We then expand on semantic atomicity specifications using two simpler examples. In Section 5.1, we discuss the effort involved in programmers writing such atomicity specifications.

We consider parallel programs in which certain regions of code are annotated as *atomic blocks*. This annotation specifies the programmers *belief* or *intention* that each atomic block is written so that, however the block is actually executed, the effect is *as if* the block's execution occurred all-at-once, with no interference or interruption from other parallel threads. For simplicity, we consider each indivisible program instruction that is not in a user-annotated atomic block to be in its own implicit, single-instruction block.

Example 1: Concurrent Queue

Consider the example program in Figure 5.1 using Java's `ConcurrentLinkedQueue` data structure, an implementation of Michael and Scott's non-blocking queue [102]. The Java class `ConcurrentLinkedQueue`, from the `java.util.concurrent` package, is implemented in a lock-free, non-blocking manner, updating its internal structure using compare-and-swap operations. If two parallel queue operations conflict, one of the operations will detect the conflict and retry.

The implementation of `ConcurrentLinkedQueue` is designed to ensure that, when multiple queue operations occur concurrently, their result is the same as if all queue operations had been executed atomically. We specify this intended atomicity in Figure 5.1 by enclosing each parallel call to `remove(1)` in a specified *atomic block*, which we write as `@assert_atomic {...}`.

```
Queue q = new ConcurrentLinkedQueue();
q.add(1);  q.add(1);

thread 1:                    thread 2:
  @assert_atomic {             @assert_atomic {
      q.remove(1);                 q.remove(1);
  }                            }

          bridge predicate:
             q.equals(q')
```

Figure 5.1: Example program with a highly-concurrent queue. The queue initially contains two copies of the value 1, and two parallel threads each try to remove a 1 from the queue. These remove operations are specified to execute as if atomic. The program is not strictly atomic, but is *semantically* atomic with respect to the given *bridge predicate*.

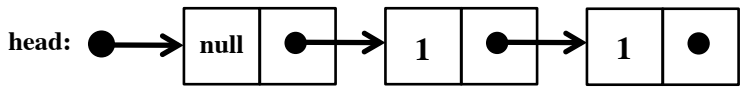

Figure 5.2: The initial internal structure of queue `q`.

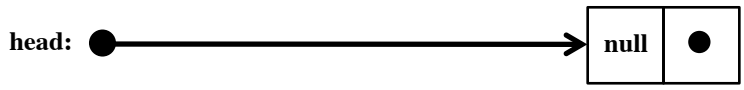

Figure 5.3: The internal structure of queue `q` after *any* serial execution of the example program.

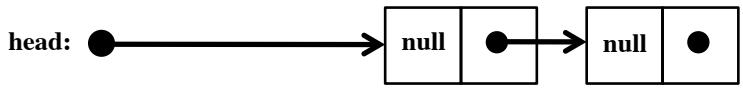

Figure 5.4: A possible internal structure of queue `q` after a parallel, interleaved execution of the example program.

If our specification was a strict atomicity specification, it would assert: for any parallel execution of the program, in which the calls to `remove(1)` can interleave, there there must exist a *serial* execution, in which each call to `remove(1)` occurred atomically, producing an *identical* final state. But this strict atomicity specification does not hold.

Internally, a `ConcurrentLinkedQueue` is a linked list. Nodes in the list can be lazily deleted — i.e. a remove operation can set the data field of a node to `null`, indicating that the corresponding data element is no longer in the queue, but leave removing the node to some future operation.

A call to the `remove`(x) method performs three basic steps: (1) Removes all nodes from the head of the list with `null` data fields, (2) Iterates through the rest of the list, searching for a node with data field equal to x, and (3) Removes x by setting the data field of the found node to `null` with a compare-and-swap. Steps (2) and (3) are repeated until the compare-and-swap succeeds or until the end of the internal list is reached.

Before the parallel threads execute `remove`, the internal list structure is as shown in Figure 5.2. In any serial execution of the two calls to `remove`, the second call will lazily delete the node `null`'ed by the first call, yielding the final internal queue structure shown in Figure 5.3. But in a parallel execution in which the two calls to `remove` are interleaved, it is possible for neither call to clean up after the other, yielding the final internal state of the queue shown in Figure 5.4.

Thus, under a traditional, strict definition of atomicity, the `remove` method is *not* atomic, as a non-serial, interleaved execution can yield a program state not reachable by any serial execution. But in either case the abstract, semantic state of queue `q` is the same — the queue is empty! That is, the code is atomic, but only at the *semantic* level of the contents of the queue.

To capture this kind of parallel correctness property, we propose *semantic atomicity*. Blocks of code are semantically atomic when, however their execution is interleaved with that of other code, their effect is *semantically equivalent* to their effect when executed serially. The desired semantic equivalence is specified by a programmer using a *bridge predicate*. For this program, semantic equivalence is given by the bridge predicate $\Phi(\sigma, \sigma')$:

$$\texttt{q.equals}(\texttt{q}')$$

This bridge predicate compares two program states, σ and σ', the first from an interleaved, parallel execution of our example program and the second from a serial execution. The unprimed `q` refers to the queue in state σ and the primed $\texttt{q}'$ refers to the queue in state σ'. This bridge predicate specifies states σ and σ' are semantically equivalent when the `equals` method of `ConcurrentLinkedQueue` returns *true* on the queues from the two states — that is, when the two queues contain the same elements, independent of their internal structure.

This semantic atomicity specification *does hold* for our example program. For any parallel execution of the program, there exists a serial execution that produces an equivalent final queue `q`.

```
int balance = 0;

void deposit1(int a) {
 @assert_atomic {
   int t = balance;
   t += a;
   balance = t;
 }
}
```

```
int balance = 0;

void deposit2(int a) {
 @assert_atomic {
   int t = balance;
   while (!CAS(&balance, t, t+a)) {
     t = balance;
   }
 }
}
```

```
int conflicts = 0;
int balance = 0;

void deposit3(int a) {
 @assert_atomic {
   int t = balance;
   while (!CAS(&balance, t, t+a)) {
     conflicts += 1;
     t = balance;
   }
 }
}

bridge predicate:
   balance == balance'
```

Figure 5.5: Three different implementations of a function to make a deposit into a bank account. Implementation `deposit1` is not atomic, while `deposit2` *is* atomic. Implementation `deposit3` is not strictly atomic, but it is *semantically* atomic with respect to the bridge predicate `balance==balance'`.

Example 2: Bank Account

Consider the code in Figure 5.5 for function `deposit1` for making a deposit into a bank account whose balance is stored in variable `balance`. The atomic specification in `deposit1` *does not hold* because there is an interleaved execution for which there is no equivalent serial execution. Suppose two threads call `deposit1`(100) in parallel, with `balance` initially 0. Under certain interleavings, both calls to `deposit1` can read a `balance` of 0 and then both can write 100 to `balance`, producing a wrong final result: `balance=100`. In contrast, any serial execution of the two threads, in which the body of `deposit1` cannot be interleaved with any other code, must produce the final result `balance=200`.

Note that atomicity violations can occur even in code that is free of data races. For example, if `deposit1` held a shared lock while reading and writing to `balance`, but released the lock in between when executing "`t += a`", then the procedure would be free of data races but would still not be atomic.

Consider instead the implementation `deposit2` in Figure 5.5, which uses an atomic compare-and-swap (`CAS`) operation to modify variable `balance`. In this implementation, the atomicity specification *does hold*. Each call to `deposit` reads `balance` into a temporary `t` and then attempts to atomically update `balance` to `t + a`, succeeding if `balance` still equals `t`. If some other thread interferes, changing `balance` between the read of `balance` and the `CAS`, then the atomic update fails and `deposit` re-reads `balance` and tries again.

Note that, while atomic, this code is not *conflict-serializable* [115]. Every parallel execution will produce the same result as a serial execution, but in a serial execution the `CAS` operation can never fail and the body of the retry loop will never be run.

Finally, consider `deposit3` in Figure 5.5. An approximate[1] count is kept, in shared

[1]Such performance counters are often not synchronized, as developers reason that the cost of synchronization is too great and approximate counts are often suitable for performance debugging.

variable `conflicts`, of the number of failed compare-and-swap operations during runs of `deposit3`. Otherwise the code is identical to `deposit2`.

Due to the introduction of shared counter `conflicts`, the atomic specification no longer holds. In a parallel execution with two calls to `deposit3(100)`, if the execution of the methods interleave, it is possible for `conflicts` to be incremented to 1. But in any execution in which the methods are executed serially, the value of `conflicts` will be 0.

The `deposit3` implementation is semantically atomic, however, w.r.t. bridge predicate:

$$\texttt{balance} == \texttt{balance}'$$

That is, if some interleaved execution produces a final `balance`, there will exist a serial execution producing a final `balance`′ such that `balance` and `balance`′ are equal.

Example 3: Multiset Stored as a List

Consider the example program in Figure 5.6, in which two threads accumulate integers into a shared `list`. If the programmer cares about the exact order of the final elements in `list`, then this code is *not* atomic. Although method `add` of `Vector` is synchronized, an interleaved execution of this example program can yield a final list of `[1,3,2,4]`, while a serial execution of the two atomic blocks could yield only `[1,2,3,4]` or `[3,4,1,2]`.

But this code can be thought of as atomic if the programmer cares only about the *multiset* of elements in the final `list`. That is, the example program in Figure 5.6 is semantically atomic with respect to some bridge predicate like `equalMultisets(list,list')`, where `equalMultisets` is a function that compares two collections to see if they have equal multisets of elements.

```
                List list = new Vector();

thread1:                          thread2:
  @assert_atomic {                  @assert_atomic {
      ...                               ...
      list.add(1);                      list.add(3);
      ...                               ...
      list.add(2);                      list.add(4);
  }                                 }

       bridge predicate:
           equalMultisets(list, list')
```

Figure 5.6: Example program in which two atomic blocks, running in parallel, insert elements into a thread-safe list.

Writing Semantic Atomicity Specs

Our above examples demonstrate that we can reap the benefits of atomicity as a specification and reasoning tool in many more contexts if we consider the more relaxed form of semantic atomicity with respect to an application-specific bridge predicate. But how much effort is involved in writing such a specification for a parallel program?

To write a semantic atomicity specification for a program, a programmer must: (1) Indicate with `@assert_atomic` which static blocks of code are intended to execute as if atomic, and (2) Write a bridge predicate to define when two final states of the program are semantically equivalent. We believe neither task should be difficult for the author of a parallel program.

We believe that, when writing multithreaded software, programmers must already be thinking about the possible interference between parallel tasks and how to prevent harmful interference using, for example, thread-safe data structures, locks, or atomic primitives such as compare-and-swaps. That is, programmers are already thinking about how to ensure that program tasks are atomic at some semantic level. Thus, it should not be difficult to specify which blocks of code are intended to behave equivalently whether or not other program tasks are run concurrently — for example, a modification to concurrent queue data structure or a deposit to a bank account. Further, in our experimental evaluation (Section 5.4), we found it to be quite simple to identify the intended atomic blocks in our benchmark programs. We specified between one and eight atomic blocks for each such benchmark.

We similarly believe that it is straightforward to specify when two program results are semantically equivalent using a bridge predicate. This specification task does not require reasoning about possible program interleavings, but simply identifying which variables or objects hold the final result of a program and considering when two such final results are semantically the same — for example, that two queues are equivalent if they contain the same elements in the same order, independent of the structure of their internal linked lists. Further, in our experimental evaluation (Section 5.4), we found that writing such bridge predicates required only a few lines of specification for each benchmark.

5.2 Semantic Atomicity and Linearizability

In this section we elaborate on several possible interpretations of atomicity specifications. We first describe these notions at a high level, and compare them to other notions of parallel correctness and non-interference. Later in this section we give the precise formal foundations on which we developed the checking algorithm described in the rest of this chapter.

Overview

At a high-level, we think of a parallel program annotated with atomic blocks as having two different execution semantics: (1) In the *interleaved* or *non-serial* semantics, the atomic

annotations have no effect — the parallel operations of different threads can be freely interleaved. (2) In the *serial* or *non-interleaved* semantics, when one program thread enters an atomic block, no other thread may execute until the first thread exits that atomic block. (Because we treat every instruction as being in an implicit single-instruction atomic block, we can think of the *serial* semantics as having a global re-entrant lock that each thread must acquire to enter any atomic block and that is released on exiting a block.)

Traditional parallel correctness properties such as *atomicity*, *serializability*, or *linearizability* hold for a program when, for any interleaved execution of the program, there exists a *similar* serial execution that produces an identical final program state. The differences between these correctness properties are in their definitions of *similar* executions:

- *Atomicity* [64] requires only that, for each interleaved execution, there exists a serial execution yielding the same final program state. The interleaved and serial executions need not be similar in any other way.

- *Serializability* [115] requires that, for each interleaved execution, there exists a serial execution which both yields the same final program state and in which all the same atomic blocks are executed.

  Conflict-serializability [115] further requires that all corresponding atomic blocks perform the same conflicting read and write operations in the interleaved and parallel execution, and that all pairs of conflicting operations occur in the same relative order. Conflict-serializability, though very strict, can be checked efficiently, and is thus used in many atomicity testing and verification tools, (e.g., [65, 64, 57, 159, 62]).

- *Linearizability* [100], like serializability, requires that, for each interleaved execution, there exists a serial execution which both yields the same final program state and in which the same atomic blocks are executed. Further, it requires that any pair of atomic blocks whose execution does not overlap in the interleaved execution must occur in the same order in the serial execution.

  Note that this definition of linearizability is somewhat different than that of [100] and later generalization [103], which formalize atomic blocks as having distinguished *responses* or *return values* and compare program states via observational equivalence — i.e. whether sequences of atomic blocks would return the same values. Our definition is appropriate for general atomic blocks without distinguished return values, while capturing the key requirement that a serial execution is equivalent only if it preserves the ordering of non-overlapping atomic blocks.

All three properties defined above require that, for each interleaved execution, there exists some serial execution producing an *identical* final state. As discussed in Section 5.1, such strict state equality is too restrictive to capture critical noninterference properties for many programs. For such programs, we propose employing a user-specified *bridge predicate* [29] — a predicate relating a pair of program states from an interleaved and a serial execution — to define a *semantic* equivalence between final program states.

That is, we can define *semantic atomicity*, *semantic serializability*, and *semantic linearizability*, all with respect to a user-specified bridge predicate Φ, by allowing in the above definitions that, for any interleaved execution with final state σ, the equivalent serial execution can have any final state σ' such that $\Phi(\sigma, \sigma')$.

The checking algorithm described in Section 5.3 tests semantic linearizability, primarily because linearizability is significantly easier to check, as it constrains the search space of similar serial executions to those that preserve the ordering of non-overlapping atomic blocks. But before describing out testing algorithm, we will briefly formalize in Section 5.2 these three parallel correctness properties.

Formal Definitions

In this section, we briefly formalize the above definitions of *semantic atomicity*, *serializability*, and *linearizability*.

Let *Thread* denote the set of program threads, Σ denote the set of program states including the thread-local states, *Atomic* denote the set of static program block *annotated* as atomic, and *Op* denote the set $\{\texttt{begin}(a) : a \in \mathit{Atomic}\} \cup \{\texttt{end}, \epsilon\}$ of atomic block operations. Intuitively, the operations are used to label the state transitions as follows. `begin(a)` marks the beginning of a dynamic instance of the static atomic block a, `end` marks the end of the last open atomic block, and ϵ is used for all other state transitions. We assume that atomic blocks are properly nested, and all instructions are inside one atomic block. If an instruction is not inside a programmer-annotated atomic block, then we assume that there is an implicit atomic block containing just that instruction.

Definition 10. *A program P consists of initial program state σ_0 and transition relation $\rightarrow$:*

$$\rightarrow \quad \subseteq \quad \Sigma \times \mathit{Thread} \times \mathit{Op} \times \Sigma$$

We write $\sigma \xrightarrow{t:\mathsf{op}} \sigma'$ when the program can transition from state σ to σ' by executing the atomic block operation `op` by thread t.

Definition 11. *An **execution** of program $P = (\rightarrow, \sigma_0)$ is a sequence of transitions in the operational semantics $\rightarrow$:*

$$\sigma_0 \xrightarrow{t_1:op_1} \sigma_1 \xrightarrow{t_2:op_2} \cdots \xrightarrow{t_n:op_n} \sigma_n$$

*An execution is **complete** if all `begin(a)` have a matching `end` operation in the same thread.*

Definition 12. *An execution E is **serial** iff, for each matched t : **begin**(a) and t : **end** transition in E, there are no transitions between the two by any other thread $s \neq t$.*

Definition 13. *A transition t : op in an execution E is a **top-level** transition if it does not occur between any matched t : **begin**(a) and t : **end** by the same thread.*

Note that a top-level transitions must be a `begin` or `end` since we assume that all instructions are part of atomic blocks.

Definition 14. *Two top-level atomic blocks* ***overlap*** *in an execution E if either of the blocks'* `begin` *or* `end` *transition occurs between the* `begin` *and* `end` *transition of the other block.*

Note that a complete execution is *serial* if and only if it contains no overlapping, top-level atomic blocks.

Definition 15. *A complete execution $E = \sigma_0 \xrightarrow{\cdots} \sigma_n$ of program P is* ***semantically linearizable with respect to*** Φ *iff there exists a serial execution $E' = \sigma_0 \xrightarrow{\cdots} \sigma'_{n'}$ of P such that:*

(1) $\Phi(\sigma_n, \sigma'_{n'})$,

(2) for every thread $t \in Thread$, the sequence of top-level operations performed by t is the same in E and E'.

(3) non-overlapping top-level atomic blocks in E appear in the same order E'.

A program P is ***semantically linearizable with respect to*** Φ *iff every execution of P is semantically linearizable.*

Note that if we remove conditions (2) and (3) above we obtain the notion of *semantic atomicity*. Similarly, if we remove only the condition (3) above we obtain the notion of *semantic serializability*. And when the bridge predicate $\Phi(\sigma, \sigma')$ is strict state equality $\sigma = \sigma'$, we obtain traditional linearizability, atomicity (removing conditions 2 and 3), and serializability (removing condition 3). We prefer to work with the stronger notion of linearizability because it is significantly easier to check.

Note that, for the purpose of checking similarity between the parallel and the serial executions, we identify the dynamic instances of atomic blocks by the combination of the thread that runs them, the static label of the atomic block, and the index of the dynamic occurrence in the execution.

5.3 Testing Semantic Linearizability

Now that we have defined semantic linearizability for programs with atomic block specifications, we can address the problem of checking the linearizability of such atomicity specifications. Suppose P is a program with atomic blocks specified to be semantically linearizable with respect to bridge predicate Φ. Checking the linearizability of P consists of two problems:

(1) Given an interleaved execution E of a program P, is E semantically linearizable with respect to Φ?

(2) Is program P semantically linearizable with respect to bridge predicate Φ? That is, is every interleaved execution of P semantically linearizable?

Given a solution to the first problem, we can in theory solve the second by enumerating all interleaved executions of P and checking if each is linearizable. In practice, however, it is typically not feasible to enumerate all executions of a parallel program.

Instead, we resort to checking the linearizability of only a subset of the interleaved executions of P. Such a checking procedure will be *sound* — if we discover any executions of P that are not linearizable, then P cannot be linearizable — but *incomplete* — even if all checked executions are linearizable, we cannot know for certain that P itself is linearizable.

There are a number of existing techniques and tools that can be applied to generate a subset of the parallel, interleaved executions of a program for testing and verification — for example, a preemption-bounded model checker [105] such as CHESS [106] or an active testing [132, 116] tool such as CalFuzzer [85]. We describe in Section 5.4 the details of our technique for generating the interleaved executions to test.

The key to the effectiveness of our approach, however, is to consider only those interleaved executions in which only a small number of atomic blocks either have their execution *interrupted* by the operations of other threads or themselves interrupt the atomic blocks of other threads.

We show in the Sections 5.3 and 5.3 that we can efficiently check the linearizability of such *interruption-bounded* executions. And in Section 5.3 we will describe a technique, leveraging optional programmer-supplied hints to further increase the efficiency of such testing. Our experimental results demonstrate that testing a program for linearizability only on *interruption-bounded* interleaved executions is sufficient to find real atomicity errors.

Interruption-Bounded Executions

Let E be an interleaved execution of some program P:

$$E \;=\; \sigma_0 \xrightarrow{t_1:op_1} \sigma_1 \xrightarrow{t_2:op_2} \cdots \xrightarrow{t_n:op_n} \sigma_n$$

We say a top-level atomic block $t : \texttt{begin}(a_i), \ldots, t : \texttt{end}$ in thread t in E is *interrupted* if any operations by other threads occur between $t : \texttt{begin}(a_i)$ and $t : \texttt{end}$ in E. The interrupting operations in the other threads are part of atomic blocks that *interrupt* the atomic block $t : \texttt{begin}(a_i), \ldots, t : \texttt{end}$.

Suppose an execution E has R interrupt*ed* atomic blocks and K interrupt*ing* atomic blocks. (Note that a single block may be both interrupted and interrupting). We ask the question, how many possible linear orderings are there of the top-level atomic blocks of E that preserve the order of non-overlapping atomic blocks in E?

We show below in Theorem 16 that such an execution E has no more than $(K+1)^R$ possible linear orderings of its top-level atomic blocks that preserve the order of non-overlapping atomic blocks. As we discuss in the next section, to check that an execution E is semantically

linearizable, we will examine linear orderings of the top-level atomic blocks of E that preserve the ordering of non-overlapping blocks in E. Thus, if execution E is *interruption-bounded* — i.e. has no more than R interrupted atomic blocks and no more than K interrupting blocks — then there will be no more than $(K+1)^R$ serial schedules that need to be examined.

Theorem 16. *Suppose an execution E has R top-level interrupted atomic blocks and K top-level interrupting atomic blocks. There are no more than $(K+1)^R$ possible linear orderings of the top-level atomic blocks of E that preserve the order of non-overlapping atomic blocks.*

Proof. The proof is by induction on R. For the base case $R = 1$, the bound is $K+1$, because the K interrupting blocks are themselves non-overlapping and thus their linear order is fixed. The interrupted block can be placed in $K+1$ positions in the order.

Suppose E has R interrupted atomic blocks and K interrupting atomic blocks. There exists some set S of $c \geq 1$ blocks in E such that: (1) every block in S is interrupted, and (2) no block in S interrupts any block not in S.

Suppose that such an S exists with $c = |S| = 1$. Then, there are no more than $(K+1)^{R-1}$ linear orderings of the remaining blocks, with the single block in S removed. And there are no more than $K+1$ ways to add back the single block into any such order, yielding the desired bound.

Suppose instead that an S exists only with $c = |S| > 1$. Then every block in S is both interrupt*ed* and interrupt*ing*. (If any block were not interrupting, then it would be an S with $c = 1$.) Thus, there are no more than $(K+1-c)^{R-c}$ linear orderings of the remaining atomic blocks, of which $R-c$ are interrupted and no more than $K-c$ are interrupting. Consider the number of distinct ways in which the c blocks of S could appear in such an ordering of the remaining blocks. There are $c!$ linear orderings of the c blocks of S. And, relative to the remaining $K-c$ interrupting blocks, there are no more than $K-c+1$ possible positions for each of the c blocks in S. Thus, the number of ways to add one linear ordering of the c blocks of S to one linear ordering of the remaining blocks is no more than $K!/c!(K-c)!$, which is the the number of ways to partition a sequence of length c into $K-c+1$ segments, allowing empty segments. The desired bound holds, as:

$$(K+1-c)^{R-c} \cdot c! \cdot \frac{K!}{c!(K-c)!} \quad \leq \quad (K+1)^R \qquad \square$$

Testing Linearizability of Interruption-Bounded Executions

Algorithm 5 lists *CheckLinearizable*(P, Φ, E), our algorithm for testing the semantic linearizability, with respect to Φ, of an execution E of a program P.

Recall that an interleaved execution E of program P is semantically linearizable w.r.t. Φ iff there exists a serial execution E' of P such that: (1) the final states of E and E' are equivalent w.r.t Φ, (2) each thread t contains the same sequence of top-level atomic blocks in E and E', and (3) if two blocks do not overlap in E, then they must occur in the same order in E'. How do we determine whether such a serial execution exists?

Algorithm 5 *CheckLinearizable*(P, Φ, E)

```
σ ← final state of execution E
for s ∈ Linearizations(E) do
  if Execute(P, s) succeeds, yielding σ' then
    if Φ(σ, σ') then
      return true
    end if
  end if
end for
return false
```

We say that a *schedule* is a sequence $(t_1, a_1), \ldots, (t_n, a_n)$ of pairs of thread identifiers t_i and atomic block labels a_i. We assume that all sources of nondeterminism in a program P, besides the scheduling of parallel threads, have been eliminated. For example, the input to P and the environment in which P runs must be fixed. Thus, the behavior of the serial executions of P, in which no atomic block interrupts the execution of any other block, are uniquely identified by the *schedule* in which the top-level atomic blocks occur.

Then, let *Linearizations*(E) be a procedure computing the set of all schedules of the top-level atomic blocks in E that preserve the order of non-overlapping atomic blocks in E. A serial execution can be a witness to the linearizability of E only if it corresponds to one of the schedules in *Linearizations*(E). By Theorem 16, the number of such schedules, and thus the number of such serial executions, is bounded by the number of interruptions in E.

We need only a mechanism for controlling the execution of a program P to force it along a schedule s. Let *Execute*(P, s) denote such a procedure. At a high level, for a program P and a schedule $s = (t_1, a_1), \ldots, (t_n, a_n)$, procedure *Execute*$(P, s)$ will, for each i from 1 to n:

- If thread t_i is not active or the next top-level atomic block to be started by t_i is not labeled a_i, then *Execute*(P, s) fails.
- Otherwise, we let thread t_i execute `begin`(a_i) and let it continue to run until it executes a matching `end`. If thread t_i blocks, *Execute*(P, s) fails. Similarly, *Execute*(P, s) fails if t_i runs forever without ever reaching a matching `end`. As the termination of t_i is undecidable, the best we can do is for *Execute*(P, s) to fail after t_i does not reach a matching `end` in a specified number of instructions.

Hints for More Efficient Testing

In testing the semantic linearizability of the atomic blocks in a program P, we expect to have to test the linearizability of many interleaved executions of P. We expect that the great majority of these tested interleavings will be linearizable — concurrency errors such as atomicity violations tend to occur only on a small fraction of executions, especially in well-tested and widely-used software. (Our experimental results match this expectation.)

If an interleaved execution E is *not* linearizable, then we will have to look at all serial ways to schedule the top-level atomic blocks of E that are consistent with the ordering of non-overlapping blocks in E. But if an interleaved execution E *is* linearizable, we can determine this fact by finding a single equivalent serial execution. This raises the possibility that, for executions that turn out to be linearizable, we could make the testing procedure described in Section 5.3 more efficient by prioritizing the order in which we examine the possible linearizations of E.

Thus, we propose two kinds of optional hints that a programmer can add to their multi-threaded code, along with their semantic atomicity specification. For an interleaved execution E, the hints will suggest which serial orderings of the overlapping atomic blocks should give equivalent results. Before falling back to a complete search of all linearizations of E, we will first try the linearizations consistent with these hints from the programmer.

Our optional hints take two forms: (1) linearization points, and (2) distinguished reads and writes.

Linearization Points

First, a user can specify *linearization points* (also called *commit points*) for atomic blocks. Any dynamic execution of an atomic block should reach at most one annotated linearization point. This hint indicates that, if two atomic blocks overlap and both execute a linearization point, then the block that executed its linearization point first should be ordered first in any serial execution.

Manually-annotated linearization points are often used in efforts to prove or verify the correctness of concurrent data structures [164, 56, 39, 153]. However, it has been observed [39, 153, 26] that it may be very difficult to identify or annotate all linearization points for some programs.

Distinguished Reads and Writes

Second, a user can annotate certain reads and writes of shared variables as *distinguished* reads and writes. When linearization points cannot be identified statically, one could often identify some distinguished shared memory accesses (i.e. reads and writes) whose ordering determines the ordering between the atomic blocks. For example, if an atomic block inserts (i.e. writes) an item to a list and another overlapping atomic block gets (i.e. reads) the same item from the list, then the ordering between the write and read accesses determines the ordering between the atomic blocks. If a CAS operation succeeds, then a shared memory write performed by the CAS is considered distinguished. On the other hand, if a CAS operation fails, then the shared memory read performed by the CAS operation can be ignored (i.e. not considered distinguished). In several of our benchmarks, we have found that even if we cannot identify the linearization points of all atomic blocks, we can identify distinguished reads and writes and use them to determine the ordering among overlapping atomic blocks. We next describe how we use distinguished operations to order atomic blocks.

Given *distinguished* operations op_1 and op_2 on the same shared variable, we say that op_1 is *ordered before* op_2 if at least one of the two operations is a write and if op_1 is executed first. Suppose atomic blocks B_1 and B_2 overlap. These hints indicate that B_1 should be ordered before B_2 in any serial execution if, for any variable v, some distinguished write to v in B_1 or the last distinguished read to v in B_1 is ordered before a distinguished write to v in B_2 or the last distinguished read of v in B_2.

These hints may indicate that B_1 should come before B_2 and that B_2 should come before B_1, in which case we ignore the distinguished reads/writes for ordering B_1 and B_2 with respect to each other.

Using Hints in Testing Linearizability

Given an interleaved execution E, we use a depth-first search to find a serial ordering consistent with the annotated hints in E. And if no execution is consistent including both the *linearization points* and the *distinguished reads and writes*, we find an ordering consistent with just the *linearization points*. We test this single serial ordering to see if it is a witness to the semantic linearizability of E, and, if not, we fall back to the exhaustive search in Section 5.3.

Our experimental results demonstrate that these hints can improve our linearizability testing to the point where the first serial execution to be examined is found to satisfy the bridge predicate. Furthermore, using these optimizations is *sound* because the testing procedure falls back to searching all other possible serial linearizations when a programmer's hints do not guide us to a witness to linearizability.

5.4 Experimental Evaluation

In this section, we describe our efforts to experimentally evaluate our approach to specifying and checking semantic atomicity for multithreaded programs. Specifically, we seek to demonstrate that:

1. We can find real atomicity errors in multithreaded programs by testing the semantic linearizability of random interleaved executions with a small number of interrupted and overlapping atomic blocks.

2. In the common case where a tested interleaving *is* linearizable, we can soundly increase the efficiency of our testing using optional programmer annotations.

Implementation

In order to evaluate our claims, we implemented our approach for Java programs. Our implementation consists of several components: (1) an annotation and assertion library for specifying which blocks of code in a Java program are intended to be semantically atomic, as

well as for specifying the bridge predicate with respect to which the blocks are intended to be atomic; (2) a component to generate random, interruption-bounded interleaved executions of a multithreaded test program; (3) a component to test the semantic linearizability of a given interleaved execution by generating and examining all serial executions that are linearizations of the interleaved execution.

Atomicity Assertion Library

Figure 5.7 shows the core API of our atomic assertion library. A programmer indicates the beginning and end of a semantic atomic block in their code by calling `Atomic.open()` and `Atomic.close()`. Each call to `open` is uniquely identified by its location in the program source (accessible by, e.g., examining a call stack trace).

The bridge predicate giving the desired semantic equivalence between interleaved and serial executions is specified via `Atomic.assert`. In an interleaved execution, a sequence of calls to `Atomic.assert(obj,pred)` indicates that there must exist some serial execution — a linearization of the interleaved execution — in which, for each corresponding call `Atomic.assert(obj',pred)`, predicate `pred.apply(obj,obj')` returns true.

That is, suppose the n^{th} call to `Atomic.assert(obj,pred)` in an interleaved execution records the serialized value of object `obj`. (We require that all objects passed to `Atomic.assert` implement the `Serializable` interface so that this recording is possible. Most common objects in the Java standard library can be serialized in this way.) Then, in a serial execution, while testing the linearizability of this interleaved execution,

```
class Atomic {

    static void open()

    static void close()

    static void assert(Object o, Predicate p)

    interface Predicate {
          boolean apply(Object a, Object b)
    }

}
```

Figure 5.7: Core atomicity specification API.

the n^{th} call to `Atomic.assert(obj',pred)` reads the previously serialized object `obj` and checks if `pred.apply(obj,obj')` holds. The serial execution is reported to be equivalent to the interleaved execution iff the same number of `Atomic.assert` calls are made and `pred.apply(obj,obj')` returns true for each one.

Sampling Interleaved Executions

Our tool for randomly generating interleaved executions of a multithreaded test program is built on top of the publicly-available and open-source CalFuzzer [85] framework for testing concurrent Java programs. CalFuzzer uses Soot [150] to instrument Java bytecode, adding calls to a user's analysis/testing code on every read, write, lock, unlock, etc. — we use these calls to take control of the parallel scheduling of a Java program and replace it with our own scheduler.

Our thread scheduler is parameterized by a maximum number R of atomic blocks to *interrupt*, a number K of other atomic blocks to execute while the atomic block is interrupted, and a bound C on the number of times to interrupt at each distinct program statement. After any statement executes in the test program, the scheduler picks the next thread to execute a statement as follows:

- If the last statement was a top-level `Atomic.close` and no thread has an open atomic block, then pick the next thread randomly from among all active threads.
- If the last statement was by thread t and thread t has an open atomic block, then subject to certain constraints, we *interrupt* the atomic block thread t is executing, selecting a random different active thread to run next.

  The constraints are: (1) We perform no more than R interruptions during an execution. (2) For each statement in each possible calling context, we interrupt at that statement only if it is in the first C occurrences in the current execution of the statement and calling context, and only if we have not interrupted at that statement, calling context, and occurrence combination in any other run.
- If the last statement was a top-level `Atomic.close` and other threads have open, interrupted blocks:

  If we have executed K complete atomic blocks since interrupting the longest-open atomic block, we select, if possible, a random active thread *in an interrupted atomic block* to run next. Otherwise, randomly select to execute next any active thread *not* in an interrupted atomic block.

Note that, barring situations in which an interrupted atomic sections becomes blocked, each interrupted atomic block will be interrupted by an expected K non-overlapping atomic blocks, although these blocks could themselves also be interrupted.

Overall, generated interleaved executions will have roughly K^R expected possible linearizations. In our experiments, we use parameters $R = 1$, $K = 4$, and $C = 4$.

Checking if an Interleaving is Linearizable

Recall from the previous section that we can use CalFuzzer [85] to control the scheduling of a parallel Java application. We use this ability to implement procedure *Execute*(P, s), described in Section 5.3, for executing a program P along a serial schedule $s = (t_1, a_1), \ldots, (t_n, a_n)$.

We then implement Algorithm 5, given this `Execute`(P, s) procedure and using the atomic assertion library described above to check the specified bridge predicate.

Benchmarks

We evaluated our approach on a number of Java benchmarks. The name, size, and number of static blocks specified as atomic is given for each of these benchmarks in Table 5.1.

The first group of benchmarks are concurrent data structures from the Java standard library `java.util.concurrent` and elsewhere. The benchmarks `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, and `CopyOnWriteArrayList` are from the Oracle Java SDK 6 (update 20). `LockFreeList` is a concurrent, lock-free list from [99], used as a benchmark by [152]. Benchmark `LazyList` is a concurrent set, implemented as a linked list with lazy deletion, from [153].

As our technique is designed to be applied to whole, closed programs, we must create a test harness for each data structure benchmark. Each harness creates one instance `obj` of the data structure and then calls four to eight methods on the instance in parallel, recording the return values. Each method call is specified to be semantically atomic with respect to a bridge predicate requiring both that `obj.equals(obj')` and that all method return values be the same.

The other group of benchmarks are from the Parallel Java (PJ) Library [86]. The PJ benchmarks include an app for computing a Monte Carlo approximation of π (`pi`), a parallel cryptographic key cracking app (`keysearch3`), an app for parallel rendering of Mandelbrot Set images (`mandelbrot`), and a parallel branch-and-bound search for optimal phylogenetic trees (`phylogeny`). Each of these benchmarks relies on roughly 15,000 lines of PJ library code for threading, synchronization, etc.

Experimental Setup and Results

For each benchmark, we execute our systematic random scheduler, described in Section 5.4 to generate a number of interleaved, *interruption-bounded* executions. We test each generated execution to see if it is semantically linearizable.

The number of *interruption-bounded* executions generated for each benchmark is listed in Column 4. Column 5 lists the number of executions found to *not* be semantically linearizable. We found non-linearizable executions for four of the data structure benchmarks and two of the PJ application benchmarks. In Column 6, we report the number of distinct bugs exposed by these atomicity-violating executions. We discuss some of these errors in detail in the

Benchmark	Approx. LoC (Benchmark + Library)	# Static Atomic Blocks	Interruption-Bounded Interleavings			Avg. # of Serial Executions			Conflicts
			total	non-linear.	errors	linear.	linear. (heuristics)	non-linear.	
`ConcurrentLinkedQueue`	200	6	241	7	2	2.96	1.20	4.29	4
`ConcurrentSkipListMap`	1400	6	487	6	2*	2.54	-	4.83	4
`ConcurrentSkipListSet`	100	6	463	5	2*	2.57	-	4.6	4
`CopyOnWriteArrayList`	600	6	222	0	0	6.23	1.0	-	0
`CopyOnWriteArraySet`	60	6	221	0	0	4.39	1.0	-	0
`LockFreeList`	100	6	319	57	1	2.08	-	3.46	2
`LazyList`	100	8	231	0	0	2.46	1.02	-	2
`PJ pi`	150 + 15,000	1	20	5	1	1.0	-	4.8	1
`PJ keysearch`	200 + 15,000	1	904	0	0	1.0	-	-	0
`PJ mandelbrot`	250 + 15,000	1	73	0	0	1.0	-	-	0
`PJ phylogeny`	4400 + 15,000	2	605	27	1	1.0	-	125.56	2

Table 5.1: Summary of our experimental evaluation of semantic atomicity specifications.

next section. Note that every violation of the linearizability of our semantic atomic blocks indicated a true error.

For the linearizable executions of each benchmark, Columns 7 and 8 report the average number of serial executions that had to be examined to find a witness to the linearizability, without and with heuristically using any hints from programmer annotations.[2] Our annotations greatly reduce the number of serial executions that must be examined for several of the data structure benchmarks.

Column 9 reports the number of serial executions examined for non-linearizable interleavings. For most benchmarks, this number is small, as expected, because we are testing the linearizability of *interruption-bounded* executions.[3]

Finally, Column 10 reports the number of atomic blocks in each data structure that are not conflict-serializable. We discuss these numbers in the next section.

Atomicity Errors Found

We now discuss several atomicity errors found by our testing.

ConcurrentLinkedQueue

Our automated testing of our semantic atomicity specification for `ConcurrentLinkedQueue` found two errors. As far as we can determine, these errors have not previously been reported.

[2]Note that, for the PJ benchmarks, because the different atomic blocks are largely independent, it is usually the case that the first serial execution we examine witnesses the linearizability.

[3]Difficulties controlling the Java scheduler benchmark `phylogeny`, however, lead to extra, unwanted interruptions and thus a larger number of serial executions that must be examined.

```
    Queue q = new ConcurrentLinkedQueue();
    q.add(1);  q.add(2);
    int sz = 0;

thread1:                        thread2:
  @assert_atomic {
      q.remove(1);                @assert_atomic {
  }                                  sz = q.size();
  @assert_atomic {                }
      q.add(3);
  }

     bridge predicate:
        q.equals(q') && (sz == sz')
```

Figure 5.8: Simple harness for `ConcurrentLinkedQueue` that reveals an atomicity error involving `add`, `remove`, and `size`. The annotated blocks are *not* semantically linearizable with respect to the bridge predicate.

The code in Figure 5.8 gives a simple test harness that exposes one of the two errors. Initially, queue `q` contains elements 1 and 2. We expect that, because methods `add`, `remove`, and `size` should all be atomic, in any parallel, interleaved execution the call to `q.size()` must return that the queue contains one element (after `q.remove(1)` but before `q.add(3)`) or two elements (before the `remove` or after the `add`). However, it is possible for `q.size()` to incorrectly report that the queue `q` contains **three** elements!

This source of this error is that computing the size of a `ConcurrentLinkedQueue` requires traversing its internal linked list structure and counting the number of elements. Suppose `thread2`'s call to `q.size()` begins its traversal, finding and counting elements 1 and 2. But, before the call sees that it is at the tail of the list, it is interrupted by `thread1`. The calls by `thread1` to `q.remove(1)` and then `q.add(3)` eliminate element 1 from the head of the list and insert element 3 at the tail of the list. Then, when `thread2`'s call to `q.size()` continues, it finds and counts element 3 and returns that queue contains three items.

Our testing found a similar error for the `toArray` method of `ConcurrentLinkedQueue`. Further, while our test harness only exercised the `add`, `remove`, `size`, and `toArray` methods of `ConcurrentLinkedQueue`, manual inspection of its source code revealed that methods `equals` and `writeObject` can similarly return non-atomic results due to iterating over the elements of the queue without checking for concurrent modifications.

We note that although `ConcurrentLinkedQueue`'s documentation[4] specifies that iteration through such a queue is only "weakly consistent", no such warning is given for methods `size`, `toArray`, `equals`, or `writeObject`. In fact, the documentation for the `size` method states:

> "Beware that, unlike in most collections, this method is NOT a constant-time operation. Because of the asynchronous nature of these queues, determining the current number of elements requires an O(n) traversal."

which seems to specify that `size`, although it requires $O(n)$ time, will return a consistent value (i.e., be linearizable). We thus judge that the unexpected behaviors of these methods are errors.

ConcurrentSkipListMap and Set

Our testing of benchmarks `ConcurrentSkipListMap` and `ConcurrentSkipListSet` found two violations of our semantic atomicity specifications for each benchmark. In particular, our test harnesses for these benchmarks each concurrently performs two insertions, two deletions, a call to `size()`, and a call to `toArray()` (or `keySet().toArray()` for `ConcurrentSkipListMap`). Our specification asserts that all six method calls execute semantically as if atomic, and our testing finds that neither method `size` nor method `toArray` is semantically atomic.

Note that the documentation[5] for `ConcurrentSkipListMap` and `ConcurrentSkipListSet` do warn for method `size`:

> "Additionally, it is possible for the size to change during execution of this method, in which case the returned result will be inaccurate. Thus, this method is typically not very useful in concurrent applications."

Thus, our specification is too strict in this case, as method `size` is not expected to be semantically atomic. Further, the documentation makes it clear that some bulk methods such as `equals`, `putAll`, etc., are not intended to be atomic. It is not clear from this documentation whether or not `toArray` is intended to be atomic.

Lock-Free List

Our automated testing of our semantic atomicity specification for the lock-free list from [99] found one previously known error. In this lock-free list, two concurrent calls to `remove` can incorrectly both report that they have successfully deleted the same single element from a list. The online errata to [99] corrects this error.

[4] `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`

[5] `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html` and `Set.html`

```
parallel-for (t in trees) {
    @assert_atomic {
        cost = compute_cost(t)
        synchronized (min_cost) {
            min_cost = min(min_cost, cost)
        }
        if (cost == min_cost) {
            min_tree = t
        }
    }
}
```

bridge predicate:

```
min_tree.equals(min_tree') && (min_cost == min_cost')
```

Figure 5.9: Simplified version of PJ `phylogeny` benchmark highlighting the nature of the atomicity error found by our technique.

PJ phylogeny Branch-and-Bound Search

We also found a previously-unknown atomicity error in Parallel Java benchmark `phylogeny`. Figure 5.9 presents a very simplified, high-level version of the benchmark that illustrates the nature of the error. Benchmark `phylogeny` is a parallel branch-and-bound search to find a minimum-cost phylogenetic tree for a given collection of DNA sequences. The search is nondeterministic — there may exist multiple minimum-cost trees and the search could return different minimum-cost trees on different runs, depending on the thread schedule and the resulting order in which the candidate trees are evaluated.

The benchmark can be thought of as parallel `for`-loop over possible phylogenetic trees. For each tree `t`, the cost is computed and the global minimum cost `min_cost` is updated. This update is safe, as proper synchronization is used to protect updates to the minimum cost. Thus, the final value of `min_cost` will always be correct.

Updates to `min_tree`, however, are not properly synchronized. Suppose one parallel loop iteration finds a new minimum-cost tree, updates `min_cost`, and enters the body of the `if`-statement with condition `cost == min_cost`, but is then interrupted by some other parallel loop iteration. The other iteration could further decrease `min_cost` and write to `min_tree`. But then when the first loop iteration continued, it would incorrectly overwrite `min_tree` with a tree no longer of minimal cost. Such an inconsistent value for `min_tree` cannot occur in a serial execution in which each parallel `for`-loop occurs atomically.[6]

[6]The error in the `phylogeny` source is that each worker thread's call to `globalResults.addAll(results)` is not atomic. This method updates the global list of minimum-cost trees with each worker thread's list of locally-minimum-cost trees.

Discussion

Comparison to Conflict-Serializability

Most existing tools for detecting atomicity violations check whether executions of a test program are conflict-serializable. As discussed in Section 5.2, conflict-serializability is a strict notion of atomicity, requiring that, for each interleaved execution, there exist a serial run in which every atomic block executes the same set of conflicting read and write operations, all in the same relative order.

Column 10 of Table 5.1 shows that several of the data structure benchmarks had atomic blocks that were not conflict-serializable, despite being semantically linearizable. Note that this column reports the number of *static* atomic blocks found to not be conflict-serializable in at least one dynamic, parallel execution. When run on all of the interruption-bounded interleavings in our experiments, a traditional atomicity analysis based on conflict-serializability would report 100+ dynamic atomicity violations for most of these benchmark. These would all be false positives (except for the few also violating semantic atomicity) that a user would have to examine. Burckhardt et al. [26] have also found that traditional atomicity analyses produce hundreds of false warnings for similar concurrent data structures.

Every atomicity violation reported by our approach, on the other hand, indicated a real atomicity violation, leading to program results not equivalent to those of any serial execution.

Effectiveness of Interruption-Bounded Testing

Our experimental results demonstrate that we can find many real semantic atomicity errors by testing linearizability on interleaved executions with a small number of interruptions. There are some errors, however, that require more interruptions to detect. For example, there is a known[7] atomicity violation in the version of Java's `ConcurrentLinkedQueue` we tested (JDK 6, update 20), involving concurrent calls to `poll()` and `remove(o)`. In particular, it is possible for a call to `remove(o)` to return true — indicating that it removed object `o` from the queue — while a parallel call to `poll()` appears to remove and return the same object `o`. But this error can occur only if the call to `remove(o)` locates `o` at the head of the queue, then `poll()` interrupts and starts to (non-atomically) remove `o` from the queue, and then `remove(o)` interrupts `poll()` and finishes its remove.

Comparison to Determinism

The error we detect in the Parallel Java (PJ) `phylogeny` benchmark was missed by our previous work [29], described in Chapter 3, which attempted to verify the semantic deterministic behavior of this and other benchmarks. For this benchmark, we previously checked a semantic deterministic specification like:

[7] `http://bugs.sun.com/view_bug.do?bug_id=6785442`

```
deterministic {
    // Phylogenetic branch-and-bound search.
    ...
} assert (min_cost == min_cost');
```

This specification asserts the following. For *any* pair of executions E and E' of this code pair from initial state σ_0 to final states σ and σ', it must be the case that `min_cost` in σ equals `min_cost` in σ'. That is, letting Φ_{det} denote bridge predicate `min_cost == min_cost`, it asserts:

$$\forall \sigma_0 \xrightarrow{E} \sigma.\ \forall \sigma_0 \xrightarrow{E'} \sigma'.\ \Phi_{\text{det}}(\sigma, \sigma')$$

To compare, let Φ_{atm} denote our bridge predicate for semantic linearizability:

$$\texttt{min\_tree.equals(min\_tree')} \ \wedge\ \texttt{(min\_cost == min\_cost')}$$

Then, our semantic atomicity specification is that, for any execution E from σ_0 to σ, there exists *at least one* serial execution E' from σ_0 to σ' such that $\Phi_{\text{atm}}(\sigma, \sigma')$ and E' is a linearization of E. That is:

$$\forall \sigma_0 \xrightarrow{E} \sigma.\ \exists \sigma_0 \xrightarrow{E'} \sigma'.\ \Phi_{\text{atm}}(\sigma, \sigma') \ \wedge\ E' \text{ a linearization of } E$$

This difference between existential and universal quantification in the specification is the core difference between the complementary notions of atomicity and determinism. Many parallel programs are intended to be deterministic — that is, to always produce semantically equivalent output for the same input, no matter the thread schedule. Deterministic specifications can exactly capture this intended determinism. But many parallel programs employ algorithms that are inherently nondeterministic and which can correctly return different final results in different runs on the same input. For example, the branch-and-bound search PJ `phylogeny`, which can correctly return different minimum-cost trees depending on the nondeterministic scheduling of its threads.

For such programs, a semantic atomicity specification can be thought of as specifying the nondeterministic behavior that is acceptable or intended — the results of any serial execution of the program, in which the atomic blocks may execute in a nondeterministic order but their executions cannot be interleaved. At the same time, the specification asserts that no additional nondeterminism — from the nondeterministic interleaving of specified atomic blocks in a parallel execution — should appear in the final results of the program. That is, the result of any interleaved execution must be semantically equivalent to the result of some serial execution.

The bridge predicate for semantic atomicity can be more strict, because it asks only: given some parallel, interleaved behavior, can the same behavior occur in *any* serial execution? But the bridge predicate for determinism specifies a kind of equivalence that must hold between any two results of correct but arbitrarily-different parallel executions.

As the final `min_tree` is *nondeterministic* — different runs can give completely different resulting trees — there is no simple way to say anything about the correctness of `min_tree`

in a deterministic assertion. On the other hand, the deterministic assertion is able to specify that no nondeterminism resulting from parallel thread scheduling should effect the final cost `min_cost`. Our semantic atomicity specifications could potentially miss errors, concurrent (perhaps if our specified atomic blocks are not large enough) or sequential, that result in incorrect final values for `min_cost` for some fraction of parallel executions.

Future Work

Our experimental results provide promising evidence both that it is feasible to write semantic atomicity specifications for parallel applications and data structures, and that we can effectively test such specifications by checking them on interruption-bounded executions. Our parallel application benchmarks are of somewhat limited size, however, and much work remains to validate our approach on a wider range of programs. In particular, we must investigate what challenges larger applications pose to writing semantic atomicity specifications and to the scalability of our testing technique.

5.5 Related Work

A large body of work has focused on verifying and testing atomicity in multithreaded programs, including static verification via type systems [65, 64], dynamic detection and prediction of atomicity violations [158, 57, 3, 159, 12, 62, 52, 37, 157], model checking of atomicity [76, 157], and active testing [132, 85, 117, 138, 91] for atomicity violations. These generally have focused on verifying that program regions are *conflict serializable* [115].

Some efforts have focused on verifying and testing notions of atomicity that are less strict than conflict-serializability of atomic sections. For example, [56] uses model checking to verify linearizability of atomic sections, but requires all atomic sections to be annotated with a linearization point which specifies the linear order in which the atomic sections must be executed in the matching serial execution. Similarly, [61] generalizes type-based verification of conflict-serializability (via reduction [93]) to account for non-conflict-serializable but side-effect-free operations like a failing compare-and-swap in a busy-waiting loop.

Another such weaker notion is *atomic-set serializability* [151], which groups storage locations into *atomic sets* and requires, for each atomic set, all atomic blocks are conflict-serializable with respect to the locations in the set. Violations of atomic-set serializability are detected dynamically by [74, 141] and found through active testing [132, 85] by [91].

A related area of research is verifying and testing linearizability[100] for concurrent objects. A concurrent object is linearizable if, for any client program which interacts with the object only through its methods and for which all such method calls are specified to be atomic, all executions of the client are linearizable. Several efforts have manually proved linearizability for certain highly-concurrent data structures [149, 39, 148]. Further, [164], [153], Line-Up [26], and CoLT [152, 133] model check linearizability of concurrent objects and libraries.

At a high level, our approach to testing semantic linearizability of atomic blocks is similar to the approach of Line-Up [26]. Line-Up generates all serial executions of a test harness using a given concurrent object, and then uses preemption-bounded model checking [105] to enumerate interleaved executions and to test that each such execution is equivalent to one of the pre-generated serial ones. Our approach, however, works on larger programs for which it is neither feasible to pre-generate all serial executions or to perform exhaustive preemption-bounded model checking, even with a small preemption bound. Rather, we randomly sample *interruption-bounded* interleaved executions and, for each such execution, examine only the corresponding serial executions (i.e. those with the same ordering of non-overlapping atomic blocks). Further, our approach is applicable to any program with annotated atomic blocks, not just concurrent objects, and our approach checks *semantic* linearizability.

Fonseca et al. developed a technique much like our semantic atomicity testing in PIKE [67], which they applied to MySQL's MyISAM storage engine — a large, core component of the MySQL database. PIKE tests the linearizability of methods of the storage engine, checking that both the values returned from each method and the final program state can be reproduced with a serial, linearized execution. But, rather than check for strict, bit-by-bit equality, Fonseca et al. write a semantic *state summary function*, which "captures an abstract notion of the state in a way that takes into consideration the semantics of the state and allows for a logical comparison, instead of a low-level physical comparison" [67]. PIKE's comparison of these abstract state summaries is analogous to evaluating one of our bridge predicates for semantic atomicity. Fonseca et al. found such semantic linearizability testing very effective for MySQL, uncovering a number of concurrency bugs, including several errors that silently (i.e. without a crash) leave the database in a state in which queries can return incorrect results.

A large body of work on *transactional memory* has developed hardware and software techniques for *implementing* atomic blocks. While such work provides hardware support, libraries, or language constructs that guarantee that blocks of code intended to be atomic are, in fact, executed atomically. Our work focuses instead on testing that a program correctly implements its intended atomicity. The kind of of semantic atomicity we test is analogous to transactional memory work on open nesting [109], transactional boosting [77], and coarse-grained transactions [87]. These lines of work achieve greater concurrency in running transactions in parallel by ignoring conflicts at the low level of reads and writes and focusing on whether data structure operations abstractly/semantically commute or conflict. For example, two calls to the `add` method of a concurrent list may conflict at both the level of individual reads and writes and when the data structure is viewed as an abstract list. But such calls can be seen to commute if the list is instead viewed as an abstract multiset.

5.6 Summary

The traditional notions of atomicity and linearizability require each interleaved execution to correspond to a serial execution that produces an identical final state. Our experiments show that the traditional interpretation of these properties is often too strong. Instead, we propose to allow the programmer to specify a bridge predicate that expresses a more relaxed, application-dependent notion of equivalence between the allowable final states.

The resulting semantic linearizability property is not only widely applicable but also effectively testable. We described and demonstrated experimentally one possible testing strategy, based on the observation that most atomicity bugs can be reproduced in parallel executions with a small number of atomic block interruptions, executions for which the set of candidate linear schedules is also small. This set of candidates can be further reduced by using programmer-annotated commit points in atomic blocks, to the point where in the common case we find the desired serial schedule on the first try. In our experiments all instances when a serial schedule could not be found were atomicity violations.

Chapter 6

NDSeq: Nondeterministic Sequential Specifications and Runtime Checking for Parallelism Correctness

In Chapters 3 and 5, we have shown that semantic determinism specifications and semantic atomicity specifications both allow us to specify critical, high-level parallelism correctness properties of multithreaded programs. But, these two specifications are only *partial* specifications of parallelism correctness.

In this chapter, we propose *nondeterministic sequential (NDSeq) specifications* [27] as a means to completely specify the correctness of a program's use of parallelism, separate from its sequential functional correctness. The key idea is for a programmer to specify the *intended* or *algorithmic* nondeterminism in a program using annotations, and then the NDSeq specification is a version of the program that is sequential but includes the annotated nondeterministic behavior. The only valid parallel behaviors are those allowed by the NDSeq specification — any *additional* nondeterminism is an error, due to unintended interference between interleaved parallel threads, such as data races or atomicity violations. Thus, a program with such annotated nondeterminism serves as its own NDSeq specification for the correctness of its parallelism.

Showing that a parallel program conforms to its NDSeq specification is a strong statement that the program's use of parallelism is correct. The behavior of the program can be understood by considering only the NDSeq version of the program, as executing the program in parallel cannot produce any different results. Testing, debugging, and verification of functional correctness can be performed on this sequential version, with no need to deal with the uncontrolled interleaving and interference of parallel threads. We show in this work that NDSeq specifications for parallel applications can be both written and checked in a simple manner, independent of an application's complex functional correctness.

In this chapter, we propose several patterns [34] for writing NDSeq specifications, and we apply these patterns to specify the parallelism correctness of a number of Java benchmarks. We find that, with a few simple constructs for specifying intended nondeterministic behavior, adding such specifications to the program text was straightforward for a variety of applications. This is despite the fact that, for many of these applications, writing a traditional functional correctness specification would be extremely difficult. (Imagine, for example, specifying the correct output of an application to render a fractal, or to compute a likely phylogenetic tree given genetic sequence data.) For many of our benchmarks, verifying that the final output is correct even for a single known input is challenging.

We propose a novel sound runtime technique [34] for checking that a structured parallel program conforms to its NDSeq specification. Given a parallel execution of such a program, we perform a conflict-serializability check to verify that the same behavior could have been produced by the NDSeq version of the program. But first, our technique combines a dynamic dependence analysis with a program's specified nondeterminism to show that conflicts involving certain operations in the trace can be soundly ignored when performing the conflict-serializability check. Our experimental results show that our runtime checking technique significantly reduces the number of false positives versus traditional conflict-serializability in checking parallel correctness.

6.1 Overview of NDSeq Specifications

In this section, we discuss an example program in detail to motivate NDSeq specifications and to informally describe our runtime checking that a parallel program is parallelized correctly with respect to its NDSeq specification. In Section 6.2, we then give a formal definition of parallelization correctness. In Section 6.3, we illustrate the generality of these specifications and of our checking approach on several examples highlighting different parallelization patterns. Section 6.4 describes the details of the runtime checking algorithm. We discuss the experimental results in Section 6.5 and conclude in Section 6.7 by pointing out significance and possible future applications of NDSeq specifications.

Motivating Example

Consider the simplified version of a generic branch-and-bound procedure given in Figure 6.1(a). This program takes as input a list of N possible solutions and computes `lowest_cost`, the minimum cost among the possible solutions, and `best_soln`, the index of a solution with minimum cost. Function `expensive_compute_cost(i)` computes the cost of solution `i`. Because this computation is expensive, the program first computes a lower bound for the cost of solution `i` with `lower_bound_cost(i)`. If this lower bound is no smaller than the lowest cost found so far (i.e. `lowest_cost`), then the program skips computing the exact cost for solution `i`.

```
 1:  coforeach (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if (b >= t)
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12:        }
13:    }
14:  }
```

(a) An example parallel search procedure.

```
 1:  nd-foreach (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if (* && (b >= t))
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12:        }
13:    }
14:  }
```

(b) Nondeterministic sequential specification for parallel search procedure.

```
 1:  coforeach (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if (true* && (b >= t))
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12:        }
13:    }
14:  }
```

(c) Parallel search with embedded nondeterministic sequential specification.

Figure 6.1: An example parallel search procedure (Figure 6.1(a)) and a nondeterministic sequential specification (NDSeq) for its parallel correctness (Figure 6.1(b)). Since a parallel program and its NDSeq specification look very similar, in practice, we do not write the NDSeq specification of a parallel program separately, but embed it in the parallel program itself. Figure 6.1(c) shows the parallel search procedure with its NDSeq specification embedded in the parallel code. The boxed constructs have different semantics when viewed as parallel code or as nondeterministic sequential specification.

The program is a *parallel* search — the **coforeach** loop allows different iterations to examine different potential solutions in parallel. Thus, updates to `lowest_cost` and `best_soln` at Lines 8–11 are enclosed in an **atomic** block, a synchronization mechanism that enforces that these lines be executed atomically — that is, all-at-once and without interruption by any other thread. Functions `expensive_compute_cost` and `lower_bound_cost` have no side-effects and do not read any mutable shared data (i.e., `lowest_cost` or `best_soln`), and thus require no synchronization.

Nondeterministic Sequential Specifications

We would like to formalize and specify that the search procedure in Figure 6.1(a) is *parallelized correctly*, and we would like some way to verify or test this parallel correctness.

If we could specify the full functional correctness of our example program — i.e. specify precisely which outputs are correct for each input — then this specification would clearly imply that the parallelization of the program was correct. But writing a full functional correctness specification is often a very difficult task. For our example search procedure, the cost of a possible solution may be a complex function whose behavior we are unable to specify, short of re-implementing the entire `expensive_compute_cost` in a specification/assertion language. Even if we could write such a specification, verifying the full functional correctness could similarly require very complex reasoning about the internals of the cost and lower bound computations.

We argue that we should seek to specify the correctness of the parallelism in our example program independently of the program's functional correctness. More generally, we aim to decompose our effort of verifying or checking the correctness of the program into two parts: (1) addressing the correctness of the parallelism, independent of the complex functional correctness and (2) addressing the functional correctness independent of any reasoning about the interleaving of parallel threads.

A natural approach to specifying parallel correctness would be to specify that the program in Figure 6.1(a) must produce the same results — i.e. compute the same `lowest_cost` and `best_soln` — as a version of the program with all parallelism removed. But if we simply replace the **coforeach** with a traditional **foreach**-loop that iterates `i` sequentially from 1 to `N`, we do not get an equivalent program. Rather, the parallel program has two "freedoms" the sequential program does not:

ND1 First, the parallel search procedure is free to execute the parallel loop iterations in any *nondeterministic* order. If there are multiple solutions of minimum cost, then different runs of the procedure may return different values for `best_soln`, depending on the order in which the loop iterations are scheduled.

The hypothetical sequential version, on the other hand, would be deterministic — it would always first consider solution 1, then solution 2, ..., up to solution `N`.

ND2 Second, the parallel program is free, in a sense, to not perform the optimization in Lines 2–5, in which the rest of an iteration is skipped because the lower bound on the solution cost is larger than the minimum cost found so far.

Consider two iterations with the same cost and with lower bounds equal to their costs. In the hypothetical sequential version, only one of the iterations would proceed to compute its cost. In the parallel code in Figure 6.1(a), however, both iterations may proceed past the check in Line 3 (as `lowest_cost` is initially ∞).

We propose to specify the correctness of the parallelism by comparing our example parallel program to a version that is sequential but contains the nondeterministic behaviors ND1 and ND2. Such a version of the program is a *nondeterministic sequential (NDSeq) specification* for the program's parallel correctness. For the program in Figure 6.1(a), our NDSeq specification is listed in Figure 6.1(b). The NDSeq specification differs from the parallel program in two ways:

1. The parallel **`coforeach`** loop at Line 1 is replaced with a sequential but nondeterministic **`nd-foreach`** loop, which can run its iterations in any order.

2. The "`* &&`" is added to the condition at Line 4. This expression "`*`" can nondeterministically evaluate to `true` or `false`, allowing the sequential specification to run the rest of a loop iteration even when `lower_bound_cost(i)` $\geq$ `lowest_cost`.

This specification is a completely sequential version of the program, executing its loop iterations one-at-a-time with no interleaving of different iterations. It contains only the controlled nondeterminism added at the two above points. We say that a parallel program *conforms* to its NDSeq specification when every final result of the parallel program can also be produced by an execution of the NDSeq specification. Section 6.2 elaborates the semantics of NDSeq specifications and our precise definition of parallelism correctness.

Note the close similarity between the parallel program in Figure 6.1(a) and its NDSeq specification in Figure 6.1(b). Rather than maintaining our parallel search procedure and its NDSeq specifications as separate artifacts, we embed the NDSeq specifications into the parallel code, as shown in Figure 6.1(c). Here we show in boxes the **`coforeach`** and "*`true`*`* &&`" to indicate that these two constructs are interpreted differently when we consider Figure 6.1(c) as a parallel program or as nondeterministic sequential one. That is, in the parallel interpretation, **`coforeach`** is a standard parallel for-loop and "*`true`*`*`" always evaluates to *true*, yielding the exact behavior of Figure 6.1(a). But when interpreted as nondeterministic sequential constructs, **`coforeach`** is treated as a **`nd-foreach`** and "*`true`*`*`" can nondeterministically evaluate to *true* or *false*, yielding exactly the behavior of Figure 6.1(b). With these annotations, the example program in Figure 6.1(c) embeds its own NDSeq specification for the correctness of its parallelism.

Runtime Checking of Parallel Correctness

We now give an overview of our proposed algorithm for the runtime checking that a parallel program conforms to its NDSeq specification. We will present the algorithm in full formal detail in Section 6.4.

Figure 6.2 illustrates a possible parallel execution of our example program in Figure 6.1(c) on $\texttt{N} = 3$ possible solutions. The three iterations of the parallel for-loop are shown in separate boxes, with the $i = 1$ iteration running in parallel with the $i = 2$ iteration, followed by the $i = 3$ iteration. Although the $i = 1$ compute a lower bound and compare it against `lowest_cost`, iteration $i = 2$ is first to compute the full cost of its solution and to update $\texttt{lowest\_cost} = 4$ and $\texttt{best\_soln} = 2$.

We would like to verify that the parallelism in this execution is correct. That is, the final result produced is also possible in an execution of the NDSeq specification. The key will be to show that parallel loop iterations together are *serializable* [115, 114] — i.e. there exist some order such that, if the iterations are executed sequentially in that order, then the same final result will be produced.

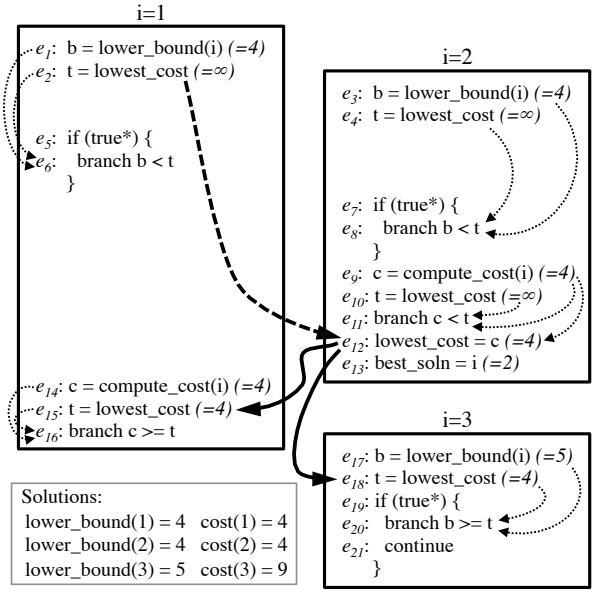

Figure 6.2: A parallel execution of three iterations (i=1,2,3) of the parallel search procedure. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thin dotted arrows denote data dependencies. The thick solid and thick dashed arrows denote transactional conflicts. Our analysis proves that the transactional conflict $e_2 \dashrightarrow e_{12}$ can be safely ignored for the serializability check.

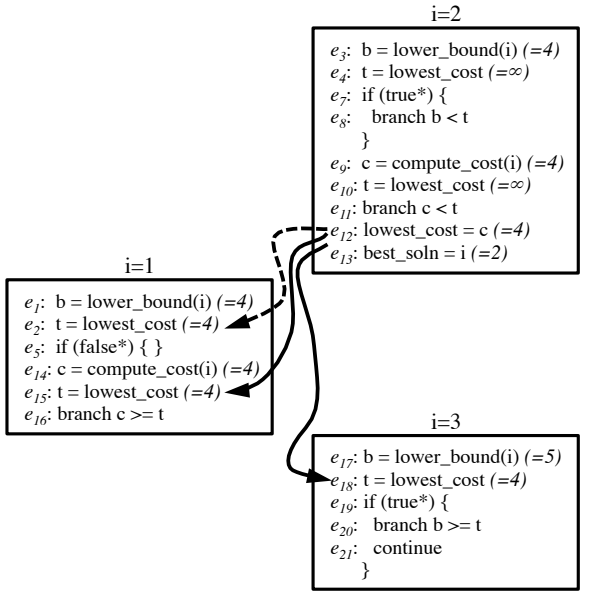

Figure 6.3: An execution of the nondeterministic sequential version of the search procedure. This execution is a serialization of the parallel execution in Figure 6.2, producing the same final result. The thick solid and thick dashed arrows denote transactional conflicts. Note that the order of conflicts $e_{12} \rightarrow e_{15}$ and $e_{12} \rightarrow e_{18}$ is the same as in Figure 6.2, while conflict $e_{12} \dashrightarrow e_2$, involving irrelevant event e_2, has been flipped.

A common restriction of serializability that can be efficiently checked and is thus often used in practice is *conflict-serializability* [115]. Given a collection of transactions — in this case, we think of each parallel loop iteration as a transaction — we form the *conflict graph* whose vertices are the transactions and with a *conflict edge* from transaction tr to tr' if tr and tr' contain conflicting operations op and op' with op happening before op'. (Two operations from different threads are conflicting if they operate on the same shared global and at least one of them is a write; in our example the conflicts are shown with thick solid or thick dashed arrows.) It is a well-known result [115] that if there are no cycles in the conflict graph, then the transactions are serializable.

But conflict-serializability is too strict for our example in Figure 6.2. There are three pairs of conflicting operations in this execution: a read-write conflict between e_2 and e_{12}, a write-read conflict between e_{12} and e_{15}, and a write-read conflict between e_{12} and e_{18}. In particular, the $i = 1$ and $i = 2$ transactions are not conflict-serializable because the $i = 2$ transaction's write of `lowest_cost` at e_{12} comes after the read of `lowest_cost` at e_2 but before the read at e_{15}.

In this chapter, we generalize conflict-serializability by determining, using a dynamic data dependence analysis, that the only use of the value read for `lowest_cost` at e_2 is in the branch condition at e_6. (The data dependence edges in each thread are shown in Figure 6.2 with the thin dashed arrows.) But because of our added nondeterminism at Line 4 of our example program, this branch condition is gated by the nondeterministic condition at e_5. Our data dependence analysis tells us that no operation performed inside the **`if(true`**$*$**`)`** opened at e_5 has any local or global side-effects — thus, in the equivalent sequential execution whose existence we are trying to show, we can choose this nondeterministic condition to be `false`, in which case the read of `lowest_cost` at e_2 will never be used. This shows that the read-write conflict involving e_2 and e_{12} is *irrelevant*, because, once we choose the nondeterministic condition to be false, the value read at e_2 has no effect on the execution.

With only the remaining two *relevant* conflicts, the conflict graph has no cycles, and thus we will conclude that the loop iterations are serializable. Figure 6.3 shows a serial execution whose existence we have inferred by verifying that there are no conflict cycles. Note that the two *relevant* conflicts, $e_{12} \rightarrow e_{15}$ and $e_{12} \rightarrow e_{18}$, are preserved — they both appear in the serial execution in the same order. But the irrelevant conflict has been flipped. The write of `lowest_cost` at e_{12} now happens before the read at e_2, but this change does not affect the final result of the execution, because the value read for `lowest_cost` does not affect the control-flow or any writes to global variables.

Now suppose that the execution in Figure 6.2 produces an incorrect result — i.e., violates a functional specification of the parallel program. Because we showed above that this execution is equivalent (with respect to the final relevant state) to the computed serial execution in Figure 6.3, then the nondeterministic sequential execution exhibits the same functional bugs as the parallel execution. Thus, we can simply debug the serial execution without worrying about thread interleavings.

6.2 Parallelism Correctness with Nondeterministic Sequential Specifications

In this section, we formally define our programming model, our nondeterministic sequential (NDSeq) specifications, and our notion of parallel correctness. As discussed in Section 6.1, we embed the NDSeq specifications for a parallel program in the program itself. We achieve this both by overloading parallel language constructs and by adding a couple of new constructs. The syntax for the language is shown in Figure 6.4.

To simplify the presentation we consider a program $\mathcal{P}$ to consist of a single procedure. We omit discussion of multiple procedures and object-oriented concepts, and we assume that each global variable refers to a distinct location on the shared heap and that each local variable refers to a distinct location on the stack of a thread. We also omit unstructured control-flow such as **`break`** and **`continue`**. Handling these details in our dynamic analysis is straightforward.

For each program $\mathcal{P}$, we define two sets of executions *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$), described below. The correctness of a parallel program is then given by relating *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$).

Parallel Executions. Set *ParExecs*($\mathcal{P}$) contains the parallel executions of $\mathcal{P}$ where each **`cobegin`** and **`coforeach`** statement implicitly creates new threads to execute its body. Statement **`cobegin`** $s_1; ...; s_n$ is evaluated by executing each of $s_1, ..., s_n$ on a separate, newly created thread. **`coforeach`** is evaluated by executing each iteration of the loop on a separate, newly created thread. Following structured fork/join parallelism, a parallel execution of a **`cobegin`** and **`coforeach`** statement terminates only after all the threads created on behalf of the statement terminate. Assignments, the evaluation of conditionals, and entire **`atomic`** statements, are executed as atomic steps without interruption by other threads. In the parallel semantics, *true*$*$ always evaluates to *true*.

$$g \in \mathit{Global} \quad l \in \mathit{Local} \quad x \in \mathit{Var} = \mathit{Global} \cup \mathit{Local}$$

$$\begin{aligned} s \in \mathit{Stmt} ::= {} & l = l \ \mathit{op}\ l \mid l = \text{constant} \mid l = l \mid g = l \mid l = g \mid s; s \\ & \mid \textbf{if}(l)\ s\ \textbf{else}\ s \mid \textbf{while}(l)\ s \mid \textbf{for}\ (l\ \textbf{in}\ l)\ s \\ & \mid \boxed{\textbf{coforeach}}\ (l\ \textbf{in}\ l)\ s \mid \boxed{\textbf{cobegin}}\ s; ...; s \\ & \mid \boxed{\textbf{atomic}}\ s \mid \boxed{\textbf{if(}\mathit{true}*\textbf{)}}\ s \end{aligned}$$

Figure 6.4: Selected statements of our language. The constructs with a different semantics in the parallel program and the sequential specification are shown in boxes.

Sequential Executions. *NdSeqExecs*($\mathcal{P}$) contains the (nondeterministic) sequential executions of $\mathcal{P}$ where all statements are evaluated sequentially by a single thread. Under the sequential semantics, the statements other than **if** with $*$, **cobegin**, and **coforeach** are interpreted in the standard way. Each evaluation of **cobegin** $s_1; ...; s_n$ is equivalent to running a nondeterministic permutation of statements $s_1, ..., s_n$, where each $s_{i\in[1..n]}$ executes sequentially. A statement **coforeach** is evaluated similarly to its deterministic version (**for**) except that the elements of the collection being iterated over are processed in a nondeterministic order. This, in essence, abstracts the semantics of the collection to an unordered set. Keyword **atomic** has no effect in the sequential case, so **atomic** s is simply equivalent to s. Finally, a *true*$*$ expression yields a nondeterministic boolean value each time it is evaluated.

Parallelism Correctness. We describe executions of $\mathcal{P}$ using a standard notion of small-step operational semantics extended with nondeterministic evaluation of **cobegin**, **coforeach**, and nondeterministic branches (i.e., **if(*true*$*$)**).

The parallelism correctness for $\mathcal{P}$ means that every final state reachable by a parallel execution of the program from a given initial state is also reachable by an NDSeq execution from the same initial state. Therefore, parallel executions have no unintended nondeterminism caused by thread interleavings: either the nondeterminism is prevented using synchronization, or it is expressed by the nondeterministic control flow in the sequential specification.

While defining parallel correctness, we distinguish a set of global variables as *focus* variables, which contain the final results of a program. Then, we reason about the equivalence of executions on the final values of the focus variables.[1]

Definition 17 (Parallelism correctness)**.** *A program $\mathcal{P}$* conforms to *its NDSeq specification with respect to a set Focus$\subseteq$ Global iff for every parallel execution $E \in$ ParExecs($\mathcal{P}$), there exists a nondeterministic sequential execution $E'\in$NdSeqExecs($\mathcal{P}$), such that the initial states of E and E' are the same and the final states agree on the values of all variables in Focus.*

6.3 Nondeterministic Specification Patterns

The use of nondeterministic sequential specifications is an attractive way to specify parallel correctness, yet it is not immediately clear where to introduce the nondeterministic constructs into the specification (1) without breaking the functional correctness while (2) capturing the nondeterminism due to thread interleavings.

Figure 6.5 shows the pseudo-code for three common patterns that we encountered repeatedly in our experiments. Each of these patterns considers parallel worker tasks where there is a potential for conflicting accesses to shared data. In contrast, applications where shared data is distributed strictly disjointly between tasks do not require use of **if(*true*$*$)** specifications. Next, we discuss these patterns in detail.

[1]Focusing only on global variables and the final states does not affect the flexibility of our method; one can make use of auxiliary global variables to keep track of valuations of local variable or the history of valuations during the course of the execution.

Optimistic Concurrent Computation

This pattern is like a manually-implemented analogue of software transactional memory (STM) [134]. A parallel task performs its work optimistically in order to reduce the synchronization with other threads. It reads the shared data (`shared`) required for its work to a local variable (`local`) and performs the computation (`do_work`) without further access to shared data. Before committing the result of the computation back to the shared space, it checks if the input it read previously has been modified by another thread (`is_conflict`). In that case it retries the work. This pattern is used when the time spent for the local computation dominates the time for checking conflict and committing, and the contention on the same regions of shared memory is low.

Such fail-retry behaviors are not normally conflict-serializable when another thread updates `shared` during `do_work`. The update conflicts with the read of `shared` before and after `do_work`. However, in those situations we expect `is_conflict` to return true and the

---

Optimistic Concurrent Computation Pattern

```
while (true) {
    local = shared;
    local' = do_work(local);
    atomic {
        if (true* && !is_conflict(local,shared)) {
            shared = local';  // commit result
            break;
        }
    }
}
```

Redundant Computation Optimization Pattern

```
if (true* && is_work_redundant(shared)) {
    // work is unnecessary; skip the work
} else {
    do_work(); // accesses shared
}
```

Irrelevant Computation Pattern

```
do_work(local, shared);
if (true*) {
    do_irrelevant_work();
}
```

Figure 6.5: Common concurrency patterns, with their expected nondeterminism specified via the *true*∗ construct.

commit to be skipped. The $\boxed{true* \;\&\&}$ allows the NDSeq program to nondeterministically skip the conflict checking and the commit. This captures that (1) it is acceptable from a partial-correctness point of view to skip the commit nondeterministically even without checking `is_conflict`, and (2) if the commit is skipped then the read of shared data before and after `do_work` are irrelevant and can be ignored for conflict serializability purposes.

Examples. This pattern is used in non-blocking data structures, e.g., stacks and queues, to implement optimistic concurrent access to the data structure without locks. These data structures implement the atomic block in the pseudo code using a compare-and-swap (CAS) operation [99]. We have also encountered this pattern when parallelizing a mesh refinement program from the Lonestar benchmark suite (see Section 6.5). Our parallelization instantiates the pattern in Figure 6.5 as follows:

```
// Processing cavity of a node N:
while (true) {
    local_cavity = read cavity of N from shared mesh;
    refined_cavity = refine the cavity locally;
    atomic {
        if (true* && mesh still contains all nodes in local_cavity) {
            replace old cavity in mesh with refined_cavity;
            break;
        }
    }
}
```

Later in this section, we give more examples of NDSeq specifications for non-blocking implementations following the same pattern.

Redundant Computation Optimization

In contrast with optimistic computation where each parallel task must complete its work, in the redundant computation pattern, each thread may choose to skip its work when it detects that the work is no longer necessary (`is_work_redundant`). Here synchronizing the check for redundancy and the actual work may not be practical when the latter is a long running operation.

Threads operating under this pattern are not conflict serializable if another thread updates the shared state while our thread calls `is_work_redundant` and finds that it returns false. Those updates conflict with the `shared` read before calling `is_work_redundant` and while executing `do_work`.

The $\boxed{true* \;\&\&}$ allows the NDSeq program to nondeterministically skip the call to `is_work_redundant` and do the work anyway. This expresses that (1) it is acceptable from a partial-correctness point of view to skip the redundancy check and to do the actual work, and also that (2) if we skip the redundancy check, then the initial read of shared state is not relevant to the computation and can be ignored for conflict-serializability purposes.

Examples. This pattern is often used when a solution space is examined by multiple threads to improve convergence to an optimal solution. Our running example in Section 6.1 follows this pattern: Lines 2–5 in Figure 6.1 test the lower bound of the current solution to decide if the computation at Line 6 can produce a better solution. The `phylogeny` benchmark from the Parallel Java Library follows a similar bound check to prune the search space for optimal phylogenetic trees. Programs using caches to avoid multiple computations of a function for the same input also use this pattern.

Irrelevant Computation

This pattern generalizes tasks that perform some computation (shown in the **`if(`*`true`*`∗)`** branch) that does not affect the rest of the execution path and does not produce a result that flows into shared state that is relevant to core functional correctness. One can ignore the irrelevant part of the program when reasoning about the program in the sequential semantics, since either (1) it does not affect the focus state, or (2) it is only necessary for the parallel executions of the program.

Examples. A prevalent instance of case (1) is when updating a statistic counter to profile the procedures of a program. For example, in the following we use **`if(`*`true`*`∗)`** to mark the increment of a statistic counter as an irrelevant operation. Updates to `counter` do not affect the final result (with respect to focus variables) of the program. However, without using the **`if(`*`true`*`∗)`**, conflicts due to `counter` will not be ignored by our analysis. By surrounding the `if` statement with **`if(`*`true`*`∗)`**, the programmer indicates that the `if` branch is not relevant for the final result, and conflicts due to the accesses to `counter` when executing the branch should not affect the parallel correctness of the program. In fact, one can easily prove statically that, given `counter` is not a focus variable, skipping the conditional at all is safe for the functionality.

```
do_work(shared, local);  // access shared variables exclusively
if (true∗) {
    if (counter < MAX_INT) {
        counter = counter + 1;
    }
}
```

Moreover, rebalancing a tree, garbage collection and compaction, maintaining a cache, and load balancing are operations that are performed to improve the performance of a parallel program, but –when implemented correctly– do not affect the core functionality of the program and thus are considered irrelevant.

An instance of (2) is when using a locking library to ensure atomicity of the relevant computation. In contrast with the statistics counters, locks are essential for the correctness of the parallelism, though the locking-state is often irrelevant for reasoning about the core functionality of the program in a sequential run.

```
1:  coforeach (i in 1,...,N) {
2:      while (true) {
3:          if ( true* ) {
4:              int prev = x;
5:              int curr = i*prev + i;
6:              if (CAS(x,prev,curr))
7:                  break;
8:          }
9:      }
10: }
11: mark_focus(x);
```

Figure 6.6: A simple parallel reduction with an embedded NDSeq specification.

Specification Example I: Non-blocking Concurrent Reduction

Consider the simple parallel program in Figure 6.6. The program consists of a parallel for-loop, denoted `coforeach` — each iteration of this loop attempts to perform a computation (Lines 4-6) based on the shared variable `x`, which is initially 0. In particular, each iteration uses an atomic compare-and-swap (`CAS`) operation to update the shared variable `x`. If multiple iterations try to concurrently update `x`, some of these `CAS`'s will fail and those parallel loop iterations will recompute their updates to `x` and then will try again.

The NDSeq specification of the program, which is embedded in the parallel program in Figure 6.6, indicates two nondeterministic aspects. First, the box around the **coforeach** construct in Line 3 specifies that the loop iterations can run in any permutation of the set `1,...,N`. This part of the specification captures the intended nondeterministic behavior of the parallel program: `x` can be updated by threads in an arbitrary order due to nondeterministic scheduling of threads. Second, the **if(*true**)** annotation in Line 3 specifies that the iteration body may be skipped nondeterministically, at least from a partial correctness point of view; this is acceptable, since the `while` loop in this program fragment is already prepared to deal with the case when the effects of an iteration are ignored following a failed `CAS` statement.

The **mark_focus** annotation in Line 11 indicates that `x` is the only focus variable. That is, the functional correctness of the program depends only on the final value of `x` after all the threads created by `coforeach` terminate.

Specification Example II: Non-blocking Concurrent Stack

In Figure 6.7, we give Java-like code for the NDSeq specification of our non-blocking **stack** benchmark. The stack is represented as a `null`-terminated, singly-linked list of `Node` objects. The head of the list is pointed by the `TOP` field of the stack. In order avoid the ABA problem, we use a version number (`TOP_version`), which is increased whenever `TOP` is updated.

The push and pop methods implement the Optimistic Concurrent Computation pattern, using a loop that iterates until the operation succeeds. Each method first reads the TOP of the stack without any synchronization and then uses an atomic CAS2 (double compare-and-swap) operation to check for conflicts by comparing TOP and TOP_version with local_top and local_v. If TOP=local_top and TOP_version=local_v then CAS2 commits the operation by writing new_top to TOP and incrementing TOP_version and returns *true*. Otherwise, CAS2 returns *false* and the operation retries.

In the specification, an **if(*true*∗)** statement is placed around the critical CAS2 in push, and around the check whether or not local_top is null in pop. These **if(*true*∗)**'s indicate that the sequential version of the program is free to nondeterministically retry as if there had been a conflict. This added nondeterminism enables our dynamic analysis to mark as irrelevant the reads of TOP and TOP_version by loop iterations in which the CAS2 fails.

The harness method creates a number of threads, each of which calls push with a random input or calls pop. In the code, we mark the focus variables using **mark_focus**. The focus variables are: (1) the values popped by the harness threads (marked after each call to pop), and (2) the last contents of the stack (marked at the end of the harness).

```
class Stack :
  Node TOP;
  int TOP_version;
}

void Stack::push(int x) {
  Node local_top, n;
  int local_v;

  n = new Node();
  n.value = x;
  while (true) {
    atomic {
      local_top = TOP;
      local_v = TOP_version;
    }
    n.next = local_top;
    if ( true* ) {
      if (CAS2(TOP, TOP_version,
               local_top, local_v,
               n, local_v+1))
          return;
    }
  }
}
```

```
class Node {
  int value;
  Node next;
}

int Stack::pop() {
  Node local_top, n;
  int local_v;

  while (true) {
    atomic {
      local_top = TOP;
      local_v = TOP_version;
    }
    if ( true* ) {
      if (local_top == null)
         return EMPTY;
      n = local_top.next;
      if (CAS2(TOP, TOP_version,
               local_top, local_v,
               n, local_v+1))
          return local_top.value;
    }
  }
}
```

```
void harness() {
  Stack stack = new Stack();
  coforeach (i = 1 .. 30) {
    // make a call to stack
    if(randomBoolean()) {
      stack.push(rand());
    } else {
      int t = stack.pop();
      mark_focus(t);
    }
  }

  // traverse the stack and
  // mark the final
  // contents relevant
  mark_focus(stack.TOP);
  Node n = stack.TOP;
  while(n != null) {
      mark_focus(n.next);
      mark_focus(n.value);
      n = n.next;
  }
}
```

Figure 6.7: A non-blocking stack implementation with an embedded NDSeq specification.

6.4 Runtime Checking of Parallel Correctness

Among the various possible techniques for checking parallelization correctness, we describe here a runtime checking algorithm. We use dynamic dataflow analysis to determine parts of the executions that are not relevant to the final valuation of the focus variables. At the same time the analysis determines appropriate assignments of boolean values to the **if(*true*∗)** nondeterministic branches in the NDSeq execution, in order to eliminate as many serialization conflicts as possible. Therefore, dataflow analysis and NDSeq specifications play key roles in improving the applicability of conflict serializability for reasoning about parallelism in real programs, for which the standard notion of serializability would be too strict. In the rest of this section, we describe the details of the checking algorithm and we sketch its correctness.

In order to simplify the presentation of the runtime checking algorithm for parallelism correctness we make the following assumptions about the parallel program being checked:

A1. Branch predicates are either *true*∗ or a local variable.

A2. The body of an **if(*true*∗)** does not contain unstructured control flow (e.g., **continue**, **break**, **return**), and there is no **else** clause.

These assumptions can be established using standard program refactoring. For example, Lines 4–5 from Figure 6.1(c) can be translated to:

```
bool cond = false;
if( true* ){ l = (b >= t); if(l){ cond = true; } }
if(cond) continue;
```

where `cond` and `l` are new local variables.

Our checking algorithm operates on an execution trace described as a sequence of execution *events*. Let τ denote a trace and e an event. For each event e we have the following information:

- *Type*(e) is the type of the event, defined as follows:

$$T \quad ::= \quad x = x' \;\mid\; branch(l) \;\mid\; branch(true*)$$

  The "$x = x'$" event type corresponds to the assignment and binary operation statements in our language (shown in Figure 6.4; recall that metavariable x stands for both locals and globals). We use a simple assignment in our formal description to simplify the presentation; unary and binary operators do not pose notable difficulties. We assume that an event can read a global, or write a global, but not both. The "*branch*(l)" event marks the execution of a branch operation when the boolean condition denoted by local l evaluates to true. The case of a branch when the negation of a local is true is similar. Finally, the "*branch*(*true*∗)" marks the execution of an **if(*true*∗)** branch, which in the parallel execution is always taken. Our algorithm does not require specific events to mark the start and end of procedures or atomic blocks.

We write $e : T$ when e has type T.

- *Thread*(e) denotes the thread that generates the event e. Recall that new threads are created when executing **cobegin** and **coforeach** statements.

- *Guard*(e) denotes the event of type *branch*(*true*$*$) that corresponds to the most recent invocation of the innermost **if(*true*$*$)** that encloses the statement generating e. For events outside any **if(*true*$*$)** this value is *nil*.

  For example, in the trace shown in Figure 6.2, *Guard*(e_6) = e_5, *Guard*(e_8) = e_7, *Guard*(e_{20}) = *Guard*(e_{21}) = e_{19}, and *Guard*(e) = *nil* for all other events e.

The checking algorithm operates in two stages, shown in Figure 6.8. The first stage computes a subset of the events in the trace that are relevant (Section 6.4), and the second stage determines whether the relevant part of the trace is conflict serializable (Section 6.4).

Selecting Relevant Events

A standard conflict-serializability algorithm [115] considers all events in a trace. We observed that in many concurrent programs it is common for partial work to be discarded when a conflict is later detected. In such cases, some of the computations based on previously read values of shared variables are discarded and are not relevant to the rest of the execution. If we can ignore such irrelevant reads of shared variables we can prove that more paths are conflict serializable. Similarly, we can ignore writes that do not affect a relevant control flow and do not flow into the final state of the focus variables. Our experiments show that this makes a difference for most benchmarks where traditional conflict serializability reports false alarms.

Informally, an assignment event is *relevant* if it targets a location that is eventually used in the computation of a final value of a focus variable, or in the computation of a *deterministic* branch. To track this relevance aspect we compute a dynamic data-dependence relation between events. For trace τ, we define the dependence relation $\dashrightarrow$ as follows:

D1. **(Intra-Thread Data Dependence).** For each local variable read $e_j : x = l$ or branch e_j : *branch*(l), we add a dependence ($e_i \dashrightarrow e_j$) on the last $e_i : l = x'$ that comes before e_j in τ. This dependence represents an actual dataflow through local l from e_i to e_j in the current trace. Both of these events are in the same thread (since they operate on the same local) and their order and dependence will be the same in any serialization of the trace. These dependence edges are shown as thin dashed arrows in Figure 6.2.

D2. **(Inter-Thread Dependence).** For each global variable read $e_j : l = g$ we add dependencies ($e_i \dashrightarrow e_j$) on events $e_i : g = l'$ as follows. From each thread we pick the last write to g that comes before e_j in τ, and the first write to g that comes after e_j in τ. This conservative dependence is necessary because the relative order of reads and writes to the global from different threads may change in a serialization of

the trace. Section 6.4 explains the importance of this detail for correctness. In the example shown in Figure 6.2, we have such dependence edges from e_{12} to all events that read `lowest_cost`: e_2, e_4, e_{10}, e_{18}.

Let $\dashrightarrow^*$ denote the transitive closure of $\dashrightarrow$.

Figure 6.8 lists the algorithm **ComputeRelevant** to compute the set of relevant events. We seed the set of relevant events with all the events that assign to the global focus variables (Line 1) and the deterministic branches outside any nondeterministic blocks (Line 2). In Line 5 we add the assignment events on which existing relevant events have data dependence.

The crucial factor that allows us to find a significant number of irrelevant events is that we can choose to skip the events (assignments and branches) corresponding to nondeterministic **`if(true*)`** blocks, as long as those blocks are irrelevant in the trace. We extend the relevance notion from assignment events to branches as follows. A *branch*(*true*$*$) event is relevant if and only if it corresponds to the execution of an **`if(true*)`** block with at least one relevant assignment event (Line 6). For this step we use the previously introduced *Guard* function to relate events inside **`if(true*)`** blocks with the corresponding *branch*(*true*$*$) event. We also say that a *branch*(l) event is relevant if it represents control flow that must be preserved (it is not nested inside an irrelevant **`if(true*)`** block). This is enforced in Lines 2 and 7. The computation in Lines 5–7 must be repeated to a fixed point since the additional relevant deterministic branches added in Line 7 can lead to new relevant assignments due to data dependencies.

For example, in the trace in Figure 6.2, relevant events are e_{9-13} from thread with $i = 2$, e_{14-16} from thread with $i = 1$, and e_{17-21} from thread with $i = 3$. Since the nondeterministic branch events e_5 and e_7 are irrelevant (no events in the rest of the trace data-depend on their bodies), the branch events e_6 and e_8 are not marked as relevant. Thus, events e_{1-2} from thread with $i = 1$ and e_{3-4} from thread with $i = 2$ have no data-dependents in *Relevant* and they remain as irrelevant.

Checking Serializability of Transactions

The final stage in our runtime checking algorithm is a conflict serializability check, implemented as cycle detection similar to [12, 62]. Our added element is that we ignore the conflicts induced by irrelevant events, and we have the flexibility to alter the nondeterministic control flow in order to remove conflicts.

First we define the conflict relation on the individual events of a trace, with respect to a subset $\mathcal{E}$ of events from the trace.

Definition 18 (Conflicting events in a set of events $\mathcal{E}$)**.** *Given a set $\mathcal{E}$ of events from a trace τ, two events $e, e' \in \tau$ are conflicting in $\mathcal{E}$ (written $e \rightsquigarrow_{\mathcal{E}} e'$) iff (a) $e, e' \in \mathcal{E}$, and (b) e occurs before e' in τ, and (c) both events operate on the same shared global variable, and at least one of them represents a write, and (d) the events are generated by different threads.*

Algorithm **ComputeRelevant**(τ, *Focus*)
Inputs: Trace τ and focus variables Focus

```
    // All writes to in-focus global variables are relevant
1   Relevant = {e : g = l ∈ τ | g ∈ Focus ∧ e is last write to g in Thread (e)}
    // All top-level, deterministic branches are relevant
2   Relevant = Relevant ∪ {e : branch(l) ∈ τ | Guard(e) = nil}
3   Compute data dependence relation -->* from τ
4   repeat
        // Add all events on which some relevant event is data-dependent
5       Relevant = Relevant ∪ {e ∈ τ | ∃e' ∈ Relevant. e -->* e'}
        // Add all nondeterministic branches enclosing a relevant event
6       Relevant = Relevant ∪ {e : branch(true*) ∈ τ | ∃e' ∈ Relevant. Guard(e') = e}
        // Add all deterministic branches enclosed inside a relevant if (true*)
7       Relevant = Relevant ∪ {e : branch(l) ∈ τ | Guard(e) ∈ Relevant}
8   until Relevant does not change
9   return Relevant
```

Algorithm **CheckCycle**(τ, *Focus*)
Inputs: Trace τ and focus variables Focus

```
    // Check serializability of trace with respect to the given NDSeq specification
10  E = ComputeRelevant(τ, Focus)
11  Compute conflict relation ⇝ between threads
12  if exists a cycle t ⇝_E t' ⇝*_E t
13      Report unserializable thread t
14  else
15      Declare the execution serializable
16  end if
```

Figure 6.8: Our two-part algorithm for checking whether or not an execution trace of a parallel program conforms to the program's nondeterministic sequential specification.

Next we lift the conflict relation from events to threads. When comparing two threads for conflicts we need to consider their events and all the events of their descendant threads. Thus, for a thread t we define its *transaction* as the set of events *Trans* (t) that includes all the events of t and of the descendant threads of t.

Definition 19 (Conflicting threads with respect to a set of events $\mathcal{E}$)**.** *Given a set $\mathcal{E}$ of events from a trace τ, two threads t, t' are conflicting in trace τ (written $t \rightsquigarrow_{\mathcal{E}} t'$) iff (a) their transaction sets are disjoint (i.e., one is not a descendant of the other), and (b) there exist two events $e \in$ Trans (t) and $e' \in$ Trans (t') that are conflicting ($e \rightsquigarrow_{\mathcal{E}} e'$). The relation $t \rightsquigarrow^*_{\mathcal{E}} t'$ is the transitive and reflexive closure of the thread conflict relation.*

The main runtime checking algorithm for parallelism correctness is shown in Lines 10–16 in Figure 6.8.

For the example trace shown in Figure 6.2 the event e_2 is not relevant, which allows the algorithm **CheckCycle** to ignore the conflict between e_2 and e_{12} (shown with thick dashed arrow in Figure 6.2). Without the dependence analysis we could not show that the trace is serializable.

Algorithm Correctness

The correctness of our runtime checking algorithm can be argued by showing that when the **CheckCycle** algorithm succeeds, the input trace $\tau \in \mathit{ParExecs}(\mathcal{P})$ can be transformed incrementally into a trace $\tau' \in \mathit{NdSeqExecs}(\mathcal{P})$ such that the final states in both traces agree on the values of the focus variables. Each incremental transformation preserves the validity of the trace and the final condition on focus variables. Some of the intermediate traces in this process will belong to the larger space of nondeterministic parallel executions $\mathit{NdParExecs}(\mathcal{P})$, which allow both interleavings (as in $\mathit{ParExecs}(\mathcal{P})$) and nondeterministic branches (as in $\mathit{NdSeqExecs}(\mathcal{P})$). For these executions nondeterministic *true*$*$ branches are resolved at runtime nondeterministically to true or false.

The first trace transformation that we perform is to eliminate the events corresponding to **if(*true*$*$)** blocks that were found irrelevant by **ComputeRelevant**. The second transformation is to commute adjacent events from different threads that are not in conflict, either because they do not operate on a shared global, or because one of them is irrelevant. The correctness of these steps — i.e., they preserve the validity of the trace and the final values of focus variables — is established by Lemma 20 and Lemma 21.

The rest of the correctness algorithm builds on a standard result from database theory: a trace is conflict serializable if it can be transformed into an equivalent serial trace by commuting adjacent, non-conflicting operations of different threads. This is possible if and only if the transactional conflict graph is acyclic [115].

Lemma 20 (Skip irrelevant nondeterministic blocks)**.** *If $\tau \in \mathit{ParExecs}(\mathcal{P})$, let τ' be the subtrace of τ obtained by eliminating all e such that $\mathit{Guard}(e) \notin$ **ComputeRelevant**(τ, Focus). Then τ' is a valid trace in $\mathit{NdParExecs}(\mathcal{P})$, meaning that τ' reflects the correct control flow of the program $\mathcal{P}$ with the corresponding irrelevant true$*$ resolved to false, and τ' agrees with τ on the final values of Focus variables. Furthermore, **ComputeRelevant**(τ', Focus) returns the same answer as for trace τ.*

The proof of this lemma relies first on the assumption (A2) stated earlier that the body of any **if(*true*$*$)** has only normal exits and no **else** clause. This means that by removing all the events in any such body results in a trace where control flows properly to the statement after the skipped **if(*true*$*$)**. All assignment events eliminated in this step are irrelevant since their guard is irrelevant (Line 6 in Figure 6.8). Therefore subsequent control flow and the final value of focus variables are preserved. The set of relevant events does not change through this transformation because its computation does not depend on irrelevant events.

Lemma 21 (Commutativity of irrelevant events)**.** *Consider a trace $\tau \in NdParExecs(\mathcal{P})$ and two adjacent events e_1 and e_2 in τ, such that the events are from different threads, they operate on a shared global g, at least one is a write event, and at least one is irrelevant — i.e., not in* ***ComputeRelevant****$(\tau, Focus)$. Then the trace obtained by commuting e_1 and e_2 is still a valid trace in $NdParExecs(\mathcal{P})$ and it agrees with τ on the final value of all of the Focus variables.*

Proof Sketch: Considering the types of events we can have in the trace and the conditions of the Lemma, we have three possible cases:

- Read-after-write: $e_1 : g = l$ and $e_2 : l' = g$. If e_2 were relevant then e_1 would also be relevant (Line 5 in the algorithm, with data-dependence rule D2). Thus it must be that e_2 is irrelevant, hence the value of l' does not affect the subsequent control flow or final values of *Focus* variables. Therefore we can commute the events and the trace remains in $NdParExecs(\mathcal{P})$. The relevant events computation does not change, since $l \neq l'$ (different threads), and the relative order of *relevant* reads and writes to global does not change.

- Write-after-read: $e_1 : l = g$ and $e_2 : g = l'$. If e_1 were relevant then e_2 would also be relevant (Line 5 in the algorithm, with data-dependence rule D2; this is a crucial part of the correctness argument that depends on the conservative form of the data-dependence rule D2). Thus, e_1 is irrelevant, and the rest of this case follows the same arguments as in the read-after-write case.

- Write-after-write: $e_1 : g = l$ and $e_2 : g = l'$. It must be that there is no nearby relevant read of the global g in the trace, or else both events would be relevant (again due to data-dependence rule D2). This means that it does not matter what we write to g. The relevant set does not change after the swap because we do not change the dependence relation $\dashrightarrow$. It is for this reason that we require the dependence rule D2 to consider the nearest write to a global from each thread. □

With these results it is straightforward to prove our main correctness result given below using standard conflict-serializability results using our relaxed notion of conflicts, as proved adequate in Lemma 21.

Theorem 22 (Correctness)**.** *Let τ be the trace generated by a parallel execution of $E \in ParExecs(\mathcal{P})$ of a program $\mathcal{P}$. If* ***CheckCycle****$(\tau, Focus)$ does not report any unserializable transaction, then there exists a nondeterministic sequential execution $E' \in NdSeqExecs(\mathcal{P})$, such that the initial states of E and E' are the same and the final states agree on the value of all variables in Focus.*

The theorem implies that if we explore, using a model checker, all parallel executions of the program and show that all these executions are serializable, then we can conclude that the parallel program conforms to its NDSeq specification.

6.5 Experimental Evaluation

In this section, we describe our efforts to experimentally evaluate our approach to specifying and checking parallel correctness using NDSeq specifications. We aim to evaluate two claims:

(1) That it is feasible to write NDSeq specifications for the parallel correctness of real Java benchmarks,

(2) Our runtime checking algorithm produces significantly fewer false positives than a traditional conflict-serializability analysis in checking parallel correctness of these benchmarks.

To evaluate these claims, we wrote NDSeq specifications for the parallel correctness of a number of Java benchmarks and then used our runtime checking technique on these specifications.

Benchmarks

We evaluate our technique on a number of benchmarks that have been used in previous research [12, 62] on parallel correctness tools, as well as in earlier chapters of this work. Note that we focus here on *parallel* applications, which use multithreading for performance but fundamentally are performing a single computation that can be understood sequentially. We do not consider *concurrent* benchmarks, such as reactive systems and stream-based systems, because it is not clear whether or not such programs can be understood sequentially.

The names and sizes of the benchmarks we used are listed in Table 6.1. Several benchmarks are from the Java Grande Forum (JGF) benchmark suite [46]: five parallel computation kernels — for successive order-relaxation (`sor`), sparse matrix-vector multiplication (`sparsematmult`), coefficients of a Fourier series (`series`), cryptography (`crypt`), and LU factorization (`lufact`) — as well as a parallel molecular dynamic simulator (`moldyn`), ray tracer (`raytracer`), and Monte Carlo stock price simulator (`montecarlo`).

Four benchmarks are from the Parallel Java (PJ) Library [86]: an app for computing a Monte Carlo approximation of π (`pi`), a parallel cryptographic key cracking app (`keysearch3`), an app for parallel rendering of Mandelbrot Set images (`mandelbrot`), and a parallel branch-and-bound search for optimal phylogenetic trees (`phylogeny`). Each of these benchmarks relies on 15K lines of PJ library code for threading, synchronization, etc.

We also applied our tool to two large benchmarks in the DaCapo benchmark suite [18], the raytracer `sunflow` and the XML to HTML converter `xalan`. Benchmark `meshrefine` is a sequential application from the Lonestar benchmark suite [88] that we have parallelized (by converting the application's main loop into a parallel loop). Benchmarks `stack` [145] and `queue` are non-blocking concurrent data structures. For each data structure, we construct a test harness that performs several insertions and removals in parallel (i.e., in a `cobegin`). The `queue` is similar to the Michael and Scott queue [102], but eagerly updates the queue's tail with a 4-word compare-and-swap. This change simplified significantly the NDSeq specification.

We report on all benchmarks that we looked at except `tsp` [122], for which we have not yet found an easy way to write the NDSeq specification (see Section 6.5).

Implementation

Although these benchmarks are written in a structured parallel style, Java does not provide structured parallelism constructs such as **`coforeach`** or **`cobegin`**. Thus, we must annotate in these benchmarks the regions of code corresponding to the bodies of parallel loops and **`cobegin`**'s. Typically, these regions are the bodies of `run` methods of subclasses of `java.lang.Thread`. Similarly, some of these benchmarks use *barrier synchronization*. As barriers have no sequential equivalent, we treat these programs as if they used a series of parallel **`coforeach`** constructs, ending one parallel loop and beginning another at each barrier. (This is a standard transformation [154] for such code.) We similarly treat each PJ benchmark, which employ sequential loops inside each of a fixed number of worker threads, as instead consisting of structured parallel loops.

In order to write NDSeq specifications, we implemented a simple library for annotating in Java programs the beginning and end of the bodies of **`if`**$(*)$, **`coforeach`**, and **`cobegin`** constructs, as well as which locations (fields, array elements, etc.) are *focus variables*. Columns 4, 5, and 6 of Table 6.1 list, for each benchmark, the number of such annotated parallel constructs, annotated **`if`**$(*)$, and statements added to mark focus variables.

We implemented our checking technique in a prototype tool for Java, which uses bytecode instrumentation via Soot [150]. In addition to the details described in Section 6.4, for Java it is necessary to handle language features such as objects, exceptions, casts, etc. Any Java bytecode instruction that can throw an exception — e.g., a field dereference, an array lookup, a cast, or a division — must be treated as an implicit branch instruction. That is, changing the values flowing into such an instruction can change the control-flow by causing or preventing an exception from being thrown.

Limitations. While our implementation supports many intricacies of the Java language, it has a couple of limitations:

- First, our implementation tracks neither the shared reads and writes nor the flow of data dependence through uninstrumented native code. Thus, we may report an execution as having correct parallelism despite unserializable conflicts appearing in calls to native code.
- Second, our tool does not instrument all of the Java standard libraries. This may cause our tool to miss data dependencies carried through the data structures in these libraries, as well as shared reads and writes inside such data structures. To address this limitation, for certain shared data structure objects we introduced fake shared variables and inserted reads or writes of those variables whenever their corresponding objects were accessed. This allows us to conservatively approximate the conflicts and data dependencies for certain critical standard Java data structures.

Benchmark		Approximate Lines of Code (App + Library)	# of Parallel Constructs	Size of Spec		Size of Trace		Distinct Serializability Warnings	
				# of `if(*)`	# of focus stmts	All Events	Irrelevant Events	Conflict-Serializability	Our Technique
JGF	`sor`	300	1	0	1	1,600k	112	0	0
	`matmult`	700	1	0	1	962k	8k	0	0
	`series`	800	1	0	5	11k	140	0	0
	`crypt`	1100	2	0	3	504k	236	0	0
	`moldyn`	1300	4	0	1	4,131k	79k	0	0
	`lufact`	1500	1	0	1	1,778k	6k	0	0
	`raytracer`	1900	1	0	1	6,170k	44k	1	1 **(b)**
	`montecarlo`	3600	1	0	1	1,897k	534k	2	0
PJ	`pi3`	150 + 15k	1	0	1	1,062k	141	0	0
	`keysearch3`	200 + 15k	2	0	4	2,059k	91k	0	0
	`mandelbrot`	250 + 15k	1	0	6	1,707k	954	0	0
	`phylogeny`	4400 + 15k	2	3	8	470k	5k	6	6 **(b)**
DaCapo	`sunflow`	24,000	4	4	3	24,250k	2,264k	28	3 **(fw)**
	`xalan`	302,000	1	3	4	16,540k	887k	6	2 **(fw)**
`stack`		40	1	2	8	1,744	536	5	0
`queue`		60	1	2	8	846	229	9	0
`meshrefine`		1000	1	2	50	747k	302k	30	0

Table 6.1: Summary of our experimental evaluation of NDSeq specifications and our runtime checking technique. The warnings for `raytracer` and `phylogeny` are all true violations, indicated with **(b)**, caused by a single bug in each benchmark. The warnings for `xalan` and `sunflow` are *false warnings*, indicated with **(fw)**.

Results: Feasibility of Writing Specifications

Writing an NDSeq specification for each benchmark program consisted of two steps: (1) adding code to mark which parts of the program's memory were *in focus* — i.e. storage locations whose values are relevant in the final program state, and (2) adding **`if`**$(*)$ constructs to specify intended or expected nondeterminism.

For all of our benchmarks besides `tsp`, it was possible to write an NDSeq specification for the benchmark's parallel correctness. The "Size of Spec" columns of Table 6.1 lists the number of **`if`**$(*)$ constructs added to each benchmark and the number of statements added to mark storage locations as in focus. These numbers show that, overall, the size of the specification written for each benchmark was reasonably small. We further found adding nondeterminism via **`if`**$(*)$ to be fairly straightforward, as all necessary nondeterminism fell under one of the NDSeq specification patterns discussed in Section 6.3. Identifying which storage locations were relevant to the final program result was similarly straightforward. As an example, we show in Section 6.3 the complete NDSeq specification for the `stack` benchmark.

Though further work is needed to evaluate the general applicability of NDSeq specifications for parallel correctness, we believe it is promising preliminary evidence that we were able to easily write such specifications for a range of parallel applications.

Results: Runtime Checking

For each benchmark, we generated five parallel executions on a single test input using a simple form of race-directed parallel fuzzing [132]. On each such execution, we checked our NDSeq specification both using our technique and using a traditional, strict conflict-serializability analysis [62].

We report in Column "Size of Trace; All Events" of Table 6.1 the size of a representative execution trace of each benchmark. The size is the number of reads and writes of shared variables and the number of branches executed during the run. Note that most of our benchmarks generate a few hundred thousand or a few million events during a typical execution on a small test input. For a dynamic analysis, this is a more relevant measure of benchmark size than static lines of code. Column "Size of Trace; Irrelevant Events" reports the number of these events found to be *irrelevant* by our algorithm **ComputeRelevant** in Figure 6.8. The fraction of events found to be irrelevant, and therefore not considered during our algorithm's serializability checking, range from 0-40%.

The "Distinct Serializability Warnings" columns of Table 6.1 report the number of serializability warnings produced by a traditional conflict-serializability check and by our technique. Note that, in a trace of a benchmark, a conflict involving a few particular lines of code may generate many cycles among the dynamic events of the trace. We report only the number of *distinct* cycles corresponding to different sets of lines of code.

Both techniques find the two real parallelism bugs — a data race in `raytracer` due to the use of the wrong lock to protect a shared checksum, and an atomicity violation in the `phylogeny` branch-and-bound search involving the global list of min-cost solutions found. (The six warnings for `phylogeny` are all real violations caused by this single bug.) Because of these bugs, neither `raytracer` nor `phylogeny` is equivalent to its NDSeq spec.

A traditional conflict-serializability analysis also gives *false warnings* for six benchmarks — incidents where there are cycles of conflicting reads and writes, but the parallel code is still equivalent to its NDSeq specification. In four of these cases, our algorithm leverages its dynamic dependence analysis and specified nondeterminism and focus variables to verify that these conflicts involve reads and writes *irrelevant* to the final program result. In this way, our algorithm eliminates all false warnings for benchmarks `montecarlo`, `stack`, `queue`, and `meshrefine`.

For benchmarks `sunflow` and `xalan`, our checking algorithm eliminates 25 and 4 false warnings, respectively, produced by strict conflict-serializability checking, but generates 3 of the same false warnings for `sunflow` and 2 for `xalan`. We discuss in greater detail below these false warnings that our analysis was and was not able to eliminate.

Note that previous work on atomicity checking, such as [62], typically evaluate on such benchmarks by checking whether or not each individual method is atomic. Thus, a single

cycle of conflicting reads and writes may lead to multiple warnings, as every method containing the cycle is reported to be non-atomic. (Conversely, multiple cycles may be reported as a single warning if they all occur inside a single method.) Our numbers of reported violations are not directly comparable, as we are interested only in whether each execution of an entire parallel construct is serializable and thus equivalent to an execution of its sequential counterpart.

`montecarlo` Benchmark. Each parallel loop iteration of the `montecarlo` benchmark contains several conflicting reads and writes on shared static fields. (The reads and writes occur inside the constructor of a temporary object created in each iteration.) To a naïve, traditional conflict-serializability analysis, these reads and writes make it appear that no equivalent serial execution exists. However, it turns out that the values written to and read from these static fields are never used — they affect neither the control flow nor the final program result. Thus, our analysis determines that these events are irrelevant and need not be considered during serializability checking. The remaining, relevant events *are* serializable, and thus our technique reports that the observed executions of `montecarlo` conform to its nondeterministic sequential specification.

`phylogeny` Benchmark. Our motivating example in Figure 6.1 is a simplified version of the branch-and-bound search `phylogeny`. Each parallel iteration of this benchmark decides whether or not to skip (or *prune*) its unit of work based on a computed lower bound and a shared `lowest_cost` variable. When an iteration does not prune its work, it may later read and possibly update the `lowest_cost` variable. This leads to violations of conflict-serializability when another thread writes to `lower_cost` in between. But this pruning check is an instance of the Redundant Computation Optimization pattern described in Section 6.3. We enclose the prune check in an **`if`**$(*)$, indicating that a sequential program is free to nondeterministically choose not to prune. Our analysis can then verify that conflicts involving the read of `lowest_cost` are irrelevant, and thus avoids the false warning a strict conflict-serializability analysis would report.

`stack`, `queue`, and `meshrefine` Benchmarks. Benchmark `meshrefine` employs the Optimistic Concurrent Computation pattern described in Section 6.3. Each parallel iteration reads from the shared triangular mesh that is being refined, and then optimistically computes a re-triangulation of a region of the mesh. It then atomically checks that no conflicting modifications have been made to the mesh during its computation and either: (1) commits its changes to the mesh if there are no conflicts, or (2) discards its optimistic computation and tries again. When conflicting modifications occur, an execution of `meshrefine` is clearly not strictly conflict-serializable. However, when we specify with an **`if`**$(*)$ that the sequential execution is free to nondeterministically discard its optimistic computation and retry, even when there are no conflicts, our analysis can verify that conflicts involving these shared reads and optimistic computation are not relevant when the optimistic work is discarded. And the remaining relevant events are serializable, so our analysis reports no false warnings here.

The `stack` and `queue` benchmarks are also instances of the Optimistic Concurrent Computation pattern, where shared reads are performed, but these reads are only relevant when a later compare-and-swap operation succeeds. Conflicts leading to failing CAS operations in these benchmarks lead to false positives for a strict conflict-serializability analysis, but our technique determines that these conflicts are not relevant to the final program result.

`sunflow` and `xalan` Benchmarks. Benchmarks `sunflow` and `xalan` cause false alarms due to the following lazy initialization pattern:

```
1:  if (true* && flag == true) {
2:      // skip initialization
3:  } else {
4:      atomic {
5:          if (flag == false) {
6:              initialize shared object
7:              flag = true;
8:          }
9:      }
10: }
```

In the above pattern, each thread checks if some shared object has been initialized and, if not, initializes the object itself. The `flag` variable, which is initially `false`, indicates whether the object has been initialized.

This pattern has two potential sources of false alarms:

1. One thread may read that `flag` is `false` at Line 1, and then another thread may initialize the object and set `flag` to `true`, so that the first thread then reads that `flag` is `true` at Line 5. This is a violation of conflict-serializability.

   This is an instance of the Redundant Computation Optimization Pattern described in Section 6.3, as a thread can always choose to skip the check at Line 1, since `flag` will be checked again at Line 5. By annotating the first check with *true*∗ &&, our analysis ignores irrelevant conflicts involving threads reading that `flag` is `false` at Line 1 and eliminates this kind of false warning.

2. When one thread initializes the shared object and sets `flag` to `true`, and then other threads read both `flag` and the shared object, our analysis sees conflict edges from the initializing thread to the other threads. These can lead to conflict cycles if the initializing thread later performs any relevant reads of data written by other threads. Our technique will report these cycles as violations.

   But such reports are false warnings, because it does not matter which thread performs this kind of lazy initialization, and it is possible to serialize such executions despite outgoing conflicts from the initialization code. Future work is needed to handle this pattern in our NDSeq specifications and dynamic checking.

Caveats

While we could easily write the specifications for our benchmarks, NDSeq specifications must be used with care. First, one must be careful to not introduce so much nondeterminism that the resulting NDSeq program exhibits undesired behavior. Our catalog of specification patterns in Section 6.3 can aid programmers in using NDSeq specifications without breaking functional correctness. Second, we note that when introducing **if(*true*∗)** one can easily introduce nontermination (as shown in several examples in this chapter). This is safe as long as we consider only the partial correctness properties of the NDSeq specification.

In order to tackle these issues, in Chapter 7 we discuss techniques for automatically inferring a *minimal* nondeterministic sequential specification such that the serializability checking described in Section 6.4 succeeds. The value of a minimal specification — i.e., with a minimal number of **if(*true*∗)**— is that it makes it more likely that the inferred specification matches the intended functionality of the program.

Our runtime algorithm reduces the number of false positives compared to a standard notion of conflict serializability. However, it can still give false positives because:

- We do not apply semantic-level analyses, such as commutativity analysis [126]. However, there are parallel patterns that rely on commutativity, in which it is necessary to ignore low-level conflicts that are part of larger, commutative operations (such as reductions). Two commutative updates on a focus variable (e.g., addition) can have conflicts yet still be serializable when the updates are considered at a semantic level. We are exploring ways to address this limitation.
- We only search for NDSeq executions for which the control-flow path outside **if(*true*∗)** branches is the same as in the parallel execution. Thus, we might miss an equivalent NDSeq path and falsely report a violation. One can eliminate this source of imprecision by exploring executions of the program with different control flows.

`tsp` Benchmark. For reasons stemming from the limitations given above, we have not found an easy way to write and check the NDSeq specification for the `tsp` [122] benchmark. Figure 6.9(a) gives the original form of the main search routine. Each thread is implemented as a loop (Lines 2–10): At each iteration it obtains a work `w` from the shared queue `Q` (Line 4), searches the region represented by `w` (Line 6) and updates the shared variable `MinTourLen` (Lines 8–9). In order to show that each thread as implemented in Figure 6.9(a) is serializable, one needs to prove that executions of `get_work` are commutative. This requires a nontrivial reasoning because procedure `get_work` may split the work items in `Q` and submit new work items to `Q`, which creates a dependency from a thread processing a work item created by a call to `get_work` by another thread. We found that Figure 6.9(a) has equivalent functionality to the rewritten form of the search in Figure 6.9(b), where each thread performs only one iteration of the `while` loop. In this case, one can show that each iteration at Lines 2–9 in Figure 6.9(b) is serializable, as procedure `recursive_solve` is a thread-local operation and the (atomic) update of `MinTourLen` at Lines 7–8 is commutative.

```
1: cobegin <1,...,N> {
2:     while (!isEmpty(Q)) {
3:         atomic {
4:             w = get_work(Q);
5:         }
6:         s = recursive_solve(w);
7:         atomic {
8:             if (s < MinTourLen)
9:                 MinTourLen = s;
10:        }
11:    }
12: }
```

(a) Original search routine

```
1: cowhile (!isEmpty(Q)) {
2:     atomic {
3:         w = get_work(Q);
4:     }
5:     s = recursive_solve(w);
6:     atomic {
7:         if (s < MinTourLen)
8:             MinTourLen = s;
9:     }
10: }
```

(b) Rewritten form of the search

Figure 6.9: Simplified version of our TSP benchmark. We were not able to easy write an NDSeq specification for the original version (a), but can write an NDSeq specification for a slightly modified version (b).

6.6 Related Work

Several generic parallel correctness criteria have been studied for shared memory parallel programs that separates the concerns about functionality and parallelism at different granularities of execution. These criteria include data-race freedom [108, 122], atomicity [64], linearizability [100]. All these criteria provides the separation between parallel and functional correctness partially, as the restriction on thread interleavings is limited, for example, to atomic block boundaries. NDSeq develops this idea up to a complete separation between parallelism and functionality so that the programmer can reason about the intended functionality by examining a sequential or nearly sequential program. NDSeq specification differs from determinism specification and checking [29, 21, 128] in that NDSeq not only allows to specify that some part of the final state is independent of the thread schedule, but also allows to specify that the part of the final state that depends on thread schedule is equivalent to the state arising due to nondeterministic choices in the NDSeq.

We formulate the checking of parallelism correctness to a general notion of atomicity. Various static [64, 149, 39, 148] and dynamic [164, 160, 97, 12, 62, 153, 37, 157, 26] techniques for checking atomicity and linearizability has been investigated in the literature. The main challenge in these techniques is to reason about conflicting accesses that are simultaneously enabled but ineffective on the rest of the execution. In the Purity work [60] Flanagan et al. provide a static analysis to rule out spurious warnings due to such conflicts by abstracting these operations to no-op's. Elmas et al. generalize this idea in a static proof system called QED [48]. They progressively transform a parallel program to an equivalent sequential program with respect to functional specifications expressed using assertions. They abstract reads and writes of shared variables; however, they need to consider functional specification when

applying the abstractions to guarantee that the abstraction does not introduce functional bug in the new program. In addition, both Purity and QED are based on Lipton's reduction theory [93], whereas we apply the idea to relax the checking of conflict-serializability [115] for nondeterministic specifications.

Atomic-set serializability [151] is an weaker notion of atomicity, which groups storage locations into *atomic sets* and requires, for each atomic set, all atomic blocks are conflict-serializable with respect to the locations in the set. Dynamic techniques for detecting violations of atomic-set serializability has been proposed [74, 91].

Recently, several systems and languages have been developed to guarantee that parallel programs give deterministic results. Techniques such as Kendo [112], DMP [42], CoreDet [15], Determinator [11], Dthreads [94], Calvin [79], and RCDC [43], employ a variety of compiler, operating system, and hardware approaches to force a multithreaded program to deterministically execute the same schedule for the same input. In the Deterministic Parallel Java (DPJ) [21] language, programmers can write parallel programs that are deterministic by construction, ensured at compile time via a static type system. DPJ also allows programmers to explicitly mark parallel constructs with nondeterministic sequential semantics and compose them safely with other deterministic constructs [22]. See Chapter 3 for a discussion of other other deterministic programming languages.

On the other hand, efforts such as the Galois project [90, 89, 118], Praun et al. [121], and Prabhu et al. [119] aim to exploit nondeterminism in irregular, sequential programs in order to efficiently parallelize them. The sequential model ensured by these systems allows nondeterministic ordering of parallel loops and pipelines. The Galois system provides a runtime with programming constructs and data structures allowing to build parallel, worklist-based applications (e.g., graph algorithms) where each iteration on the worklist is treated a coarse grained transaction. The system performs conflict detection at the object level and based on the semantics of the object and performs rollback upon unspecified conflicts. Similarly, ALTER [147] has programmers annotate loops which may be executed nondeterministically and shared reads which may nondeterministically read stale values, and uses these annotations to automatically parallelize sequential programs. The resulting program uses STM-style techniques to detect conflicts, to respect the sequential semantics, and to achieve deterministic execution. Praun et al. [121] propose the programming model IPOT that allows programmers to explicitly mark portions of the program for speculative multithreaded and transactional execution. Its `tryasync` construct resembles our `cobegin` construct. IPOT allows internal nondeterminism in that intermediate states may differ from the corresponding sequential execution, but guarantees external determinism where the final state only depends on the inputs, not the thread interleavings. Their runtime technique aims to preserve, rather than checking, sequential semantics. Prabhu et al. [119] propose speculative composition and iteration as programming constructs to parallelize parts of the program with explicit dependencies. They guarantee the obedience to sequential semantics by running a sequential version of the program that verifies the speculated values of each parallel part. Saltz et al. [129], and Rauchwerger et al. [125] present runtime checks for parallelizing executions of loops. Their runtime techniques complement static transformations by tracking at runtime

data dependencies across parallel loop iterations similarly to our runtime algorithm does to identify true conflicts between threads. HAWKEYE [146] similarly is a dynamic dependence analysis, for identifying data dependencies that prevent parallelization (i.e., "dataflow impediments" to achieving identical results with a parallelized version). But HAWKEYE can compute dependencies at the semantic level of a program's abstract data structures, taking commutativity into account, rather than only examining dependencies at the low-level of individual memory accesses.

6.7 Summary

We proposed the use of nondeterministic sequential specifications to separate functional correctness from parallelism correctness of parallel programs. Our proposal has several advantages.

First, NDSeq specifications are lightweight. Unlike traditional mechanisms for functional specification, e.g., invariants and pre/post-conditions, NDSeq specifications do not require one to learn a complex logic or language. The original parallel program along with a few **`if(`*`true`*`*)`** serves as the specification and can be used alone to detect various parallelism-related bugs.

Second, once we verify parallelism correctness, proving the functional partial correctness of the parallel program amounts to checking the functional correctness of the NDSeq program. Threads being absent, this can be done using well-developed techniques for verifying sequential programs. Note that verification of sequential programs (even with nondeterminism) is much simpler than verification of parallel programs. For example, model checking of Boolean multithreaded programs is undecidable [123], whereas model checking of Boolean nondeterministic sequential programs is decidable [53]. The latter fact has been exploited by several well-known model checkers for nondeterministic sequential programs [13, 36, 17]. Similarly, NDSeq specifications also simplify the reasoning about other concurrency-related properties such as determinism and linearizability.

Third, NDSeq specifications can simplify debugging of functional correctness bugs. When investigating a parallel execution that exhibits a bug, the programmer can be presented with the equivalent, hence similarly buggy, NDSeq execution. This allows the programmer to analyze the bug by examining a sequential behavior of the program, which is much easier to debug than its parallel counterpart.

We proposed a runtime checking algorithm for parallelism correctness. Our algorithm is based on a combination of simple dynamic dataflow analysis and conflict serializability checking. The NDSeq specification is the key factor that improves the precision of conflict serializability by indicating the conflicts that can be safely be ignored by the analysis. We believe that a similar verification can be done statically; such an extension remains a future work. A key aspect of our checking algorithm (unlike static proof systems [48] and type systems [64]) is that it does not need to refer to functional invariants which often complicates the verification process.

Chapter 7

NDetermin: Inferring Likely Nondeterministic Sequential Specifications for Parallelism Correctness

In Chapter 6, we proposed *nondeterministic sequential (NDSeq) specifications* [27, 34] as a way to separately address the correctness of a program's parallelism and a program's sequential functional correctness. The key idea is to use a sequential version of a structured-parallel program as a specification for the program's parallel behavior. That is, the program's use of parallelism is correct if, for every parallel execution of the program, their exists an execution of the sequential version that produces the same result. But, to capture all intended behaviors of a parallel program, the sequential version may have to include some limited, controlled nondeterminism. Thus, a programmer specifies the *intended* or *algorithmic* nondeterminism in their parallel application, and the nondeterministic sequential (NDSeq) specification is a sequential version of the program, with no interleaving of parallel threads but with this annotated nondeterminism. Any *additional* nondeterminism is an error, due to unintended interference between interleaved parallel threads, such as data races or atomicity violations. Further, the functional correctness of a parallel program can then be tested, debugged, and verified *sequentially* on the NDSeq specification, without any need to reason about the uncontrolled interleaving of parallel threads.

We also proposed in Chapter 6 a sound runtime technique [34] for checking that a structured-parallel program conforms to its NDSeq specification. The technique generalizes traditional conflict-serializability checking, by using a dynamic dataflow an analysis and the NDSeq specification's nondeterminism to safely ignore irrelevant conflicting program operations. This technique was able to check the parallelism correctness of a number of complex Java benchmark programs. However, writing NDSeq specifications for parallel programs can be a time-consuming and challenging process, especially to a programmer unfamiliar with such specifications. A programmer can easily forget to include some correct nondeterministic

behaviors, forcing them to iterate between checking their program against its NDSeq specification and inspecting violating executions in order to add such missing nondeterminism to the specification.

While NDSeq specifications enable one to check serializability properties of parallel programs where traditional conflict-serializability checking fails, the technique given in Chapter 6 is not fully automatic — it requires the user to provide an NDSeq specification of the parallel program. Although we provides in Chapter 6 a list of patterns for writing NDSeq specifications in common cases, we observe that coming up with a NDSeq specification for a parallel program can be tedious and time-consuming for programs with complex synchronization and communication. The user has to follow a manual approach where he or she observes a few parallel execution traces of the program, spends some time to analyze them, and then comes up with a plausible NDSeq specification. After producing a first NDSeq specification, the user runs the proposed runtime checking algorithm from Chapter 6. If the checking algorithm finds any violation of the specification, the user needs to go back to analyze the trace that violated the specification and try to manually verify if the trace contains a real parallel bug. If not, the user has to try to improve the specification with insights obtained from the analysis of the failing trace. Although this iterative approach can be effective in finding NDSeq specifications for small benchmarks, for large and complex programs it can be highly time consuming and tedious.

In this chapter, we propose a technique [33] for automatically inferring a likely NDSeq for a structured-parallel program. Given a representative set of correct parallel executions, plus some simple annotations about which program locations contain the final program result, our algorithm infers a NDSeq specification with a minimal amount of added nondeterminism necessary to capture all behavior seen in the observed parallel executions. Our inference algorithm combines dynamic dataflow analysis, conflict-serializability checking, and Minimum-Cost Boolean Satisfiability (MinCostSAT) solving.

We have implemented our NDSeq specification inference algorithm in a prototype tool for Java, called nDetermin. We applied nDetermin to the Java benchmarks from Chapter 6 for which we previously hand-wrote NDSeq specifications, and nDetermin correctly inferred all the hand-written specifications. This provides promising preliminary evidence that nDetermin can infer correct and useful NDSeq specifications for parallel applications.

We believe that automatically inferring NDSeq specifications can save programmer time and effort in applying NDSeq specifications. In particular, using an inferred specification as a starting point is much simpler than writing a whole specification from scratch. Further, our inference algorithm can detect parallel behaviors that *no* possible NDSeq specification would allow, which often contain parallelism bugs. More generally, inferred specifications can aid in understanding and documenting a program's parallel behavior. Finally, inferring NDSeq specs is a step towards an automated approach to testing and verification of parallel programs by decomposing parallelism and sequential functional correctness — where a program's parallelism would be checked against its inferred NDSeq spec, while functional correctness would be checked *sequentially* on the NDSeq specification using any of a wide variety of powerful techniques for testing and verifying sequential, nondeterministic programs.

7.1 Overview

In this section, we give an overview of our algorithm for inferring NDSeq specifications for parallel programs on a simple example. To provide context, we also recap some background material from Chapter 6 on writing and checking NDSeq specifications.

Running Example

Consider the simple parallel program in Figure 7.1. The program consists of a parallel for-loop, written as `coforeach` — each iteration of this loop attempts to perform a computation (Line 6) on shared variable `x`, which is initially 0. Each iteration uses an atomic compare-and-swap (`CAS`) operation to update shared variable `x`. If multiple iterations try to concurrently update `x`, some of these `CAS`'s will fail and those parallel loop iterations will recompute their updates to `x` and try again.

Consider the parallel execution shown in Figure 7.3. In this execution, the `i=1` iteration reads and computes an updated value for shared variable `x`. But before the `i=1` iteration can update `x`, the `i=2` iteration (in another thread) runs and sets `x` to 2. The first compare-and-swap (`CAS`) operation in the `i=1` iteration then fails, and the iteration redoes its computation before successfully updating `x`.

Background: Nondeterministic Sequential Specifications

Instead of testing or verifying a parallel program directly against a functional specification, we would like to separate this end-to-end reasoning into two simpler tasks: (1) the checking of whether the program is *parallelized correctly* independent of the complex functional correctness, and (2) the checking of whether the program satisfies a functional correctness criteria independent of any interleaving of threads.

A natural approach to specifying parallelism correctness would be to specify that the program in Figure 7.1 must produce the same final value for `x` as a version of the program with all parallelism removed — i.e., the entire code is executed by a single thread. (Note that, this condition is independent of which final values of `x` are correct, which is specified and checked separately as the functional correctness of the program.) However, in this case we do not get a sequential program equivalent to the parallel program. For example, the parallel program in Figure 7.1 is free to execute the computations at Line 6 in any *nondeterministic* order. Thus, for the same input value of `x`, different thread schedules can produce different values for `x` at the end of the execution. On the other hand, executing the loop sequentially from `1` to `N` will always produce the same, deterministic final value for `x`. Suppose that such extra nondeterministic behaviors due to thread interleavings are intended; the challenge here is how to express these nondeterministic behaviors in a sequential specification.

In Chapter 6, we addressed this challenge by introducing a specification mechanism that the programmer can use to declare the intended, algorithmic notions of nondeterminism in the form of a sequential program. Such a *nondeterministic sequential specification* (NDSeq)

```
 1: coforeach (i in 1,...,N) {
 2:    bool done = false;
 3:    while (!done) {
 4:
 5:       int prev = x;
 6:       int curr = i * prev + i;
 7:       bool c = CAS(x, prev, curr);
 8:       if (c) {
 9:          done = true;
10:       }
11:
12:    }
13: }
```

Figure 7.1: A simple parallel reduction on {1,...,N}, done in arbitrary order.

```
 1: nd-foreach (i in 1,...,N) {
 2:    bool done = false;
 3:    while (!done) {
 4:      if (*) {
 5:        int prev = x;
 6:        int curr = i * prev + i;
 7:        bool c = CAS(x, prev, curr);
 8:        if (c) {
 9:           done = true;
10:        }
11:      }
12:    }
13: }
```

Figure 7.2: A nondeterministic sequential specification for the program in Figure 7.1.

for our example program is shown in Figure 7.2. This specification is intentionally very close to the actual parallel program, but its semantics are sequential with two nondeterministic aspects. First, the **nd-foreach** keyword in Line 1 specifies that the loop iterations can run in any permutation of the set 1, ..., N. This part of the specification captures the intended nondeterministic behavior of the program, caused in the parallel program by running threads with arbitrary schedules. Second, the **if(*true*∗)** keyword in Line 4 specifies that the iteration body may be skipped nondeterministically, at least from a partial correctness point of view; this is acceptable, since the loop in this program fragment is already prepared to deal with the case when the effects of an iteration are ignored following a failed CAS statement. In summary, all the final values of `x` output by the parallel program in Figure 7.1 can be produced by a feasible execution of the NDSeq specification in Figure 7.2. Then, we say that the parallel program obeys its NDSeq specification. In fact, there exists a sound algorithm [34] that checks for a given representative interleaved execution trace τ of the parallel program, whether there exists an equivalent, feasible execution of the NDSeq specification.

Inferring NDSeq Specifications

The key difficultly with the previous approach is that writing such specifications, and especially the placement of the **if(*true*∗)** constructs, can be difficult in many practical situations. If we place too few **if(*true*∗)** constructs, we may not be able to specify some intended nondeterministic behaviors in the parallel code. However, if we place too many **if(*true*∗)** constructs, or if we place them in the wrong places, the specification might allow too much nondeterminism, which will likely violate the intended functionality of the code.

Our contribution in this chapter is to give an algorithm, running on a set of input execution traces, for inferring a *minimal* nondeterministic sequential specification such that

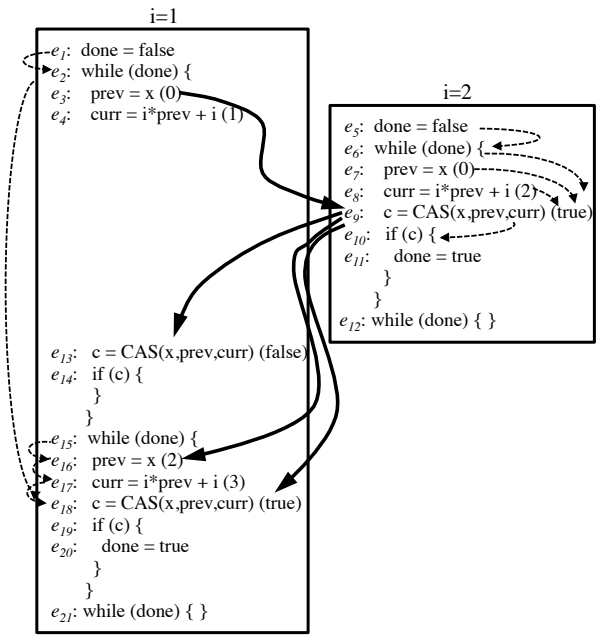

Figure 7.3: A parallel execution of two iterations (i=1,2) of the example parallel program from Figure 7.1. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thin dotted arrows denote data dependencies between events. The thick solid arrows denote transactional conflicts.

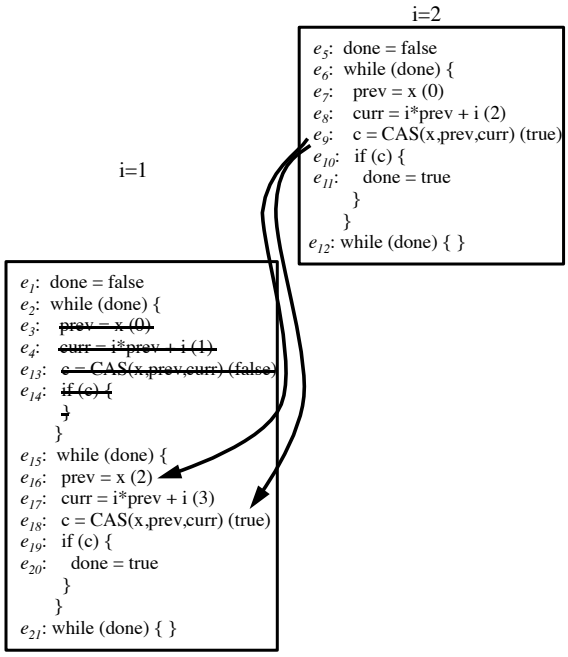

Figure 7.4: An equivalent serialization of the parallel execution in Figure 7.3. This serialization is an exection of the NDSeq specification in Figure 7.2. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thick solid arrows denote transactional conflicts.

the checking approach described in Chapter 6 suceeds on the input traces. Choosing a minimal specification — i.e., with a minimal number of **`if(true*)`**, is a heuristic that makes it more likely that the inferred specification matches the intended behavior of the program. Our key idea is to reformulate the runtime checking algorithm (Figure 6.8) of Chapter 6 as a constraint solving and optimization problem, in particular a Minimum Cost Boolean Satisfiability (MinCostSAT) problem.

Runtime Checking Parallelism Correctness. Consider the parallel execution shown in Figure 7.3. The algorithm from Chapter 6 checks if this trace can be serialized with respect to the NDSeq specification — i.e. whether the final result (the value of the shared variable `x`) can be obtained by running the loop iterations sequentially in some nondeterministic order. For our example trace, the algorithm discovers the serialization in Figure 7.4. This serialization is a *witness* to the correctness of the parallelism in the trace in Figure 7.3.

The algorithm in Chapter 6 determines whether or not such an NDSeq execution exists

by generalizing *conflict-serializability* [114] checking. We now describe conflict-serializability on our motivating example to show why and how conflict-serializability must be generalized for checking NDSeq specifications. Given a collection of transactions — in this case, we think of each parallel loop iteration as a transaction — we form the *conflict graph* whose vertices are the transactions and with a *conflict edge* from transaction tr to tr' if tr and tr' contain conflicting operations op and op' with op happening before op'. Two operations from different threads are *conflicting* if they operate on the same shared global and at least one of them is a write; in Figure 7.3 and Figure 7.4 the conflicts are shown with thick solid arrows. It is a well-known result [114] that if there are no cycles in the conflict graph, then the transactions are serializable.

Because the conflict arrows from the `i=1` iteration to the `i=2` iteration (from e_3 to e_9) and vice versa (e.g., from e_9 to e_{13}, e_{16}, or e_{18}) form a cycle, these two iterations are not *conflict-serializable*. Yet, this execution trace is serializable, since its result is the same as if we run first the iteration `i=2` followed by `i=1`. Therefore, we need a more general notion than conflict-serializability.

In order to report this execution serializable, we must be able to show that all conflict cycles between iteration `i=1` and `i=2` can be safely ignored. Assuming that we are given the specification in Figure 7.2, we give next the basic reasoning behind how the NDSeq specification is used to ignore the conflict cycles in the parallel execution. Then, we explain how we encode this reasoning in a MinCostSAT formula in order to infer the NDSeq specification that will enable us to ignore the conflict cycles. For this, we perform a dynamic dataflow analysis and use the **`if(true*)`** in the program's NDSeq specification in this analysis. In particular, we need to identify *relevant* events in the traces: (1) final writes to the shared variable `x`, and (2) all events on which events in (1) are (transitively) *dependent*. Then, we check if there is any conflict cycle formed by only relevant events; we can safely ignore the cycles that contain irrelevant events.

Computing Relevant Events. When computing the set of relevant events, we consider all data dependencies between events (shown with thin dotted arrows in Figure 7.3). For the trace in Figure 7.3, we first include events e_9 and e_{18} in the relevant events, as both write to shared variable `x`. We then include e_7, e_8, e_{16}, and e_{17}, as e_9 and e_{18} are *data dependent* on these events.

The way we consider control dependencies is subtle. By default, a deterministic branch event is considered relevant and all events that flow into its branch condition become relevant. For example, in Figure 7.3, the events e_2, e_6, e_{10}, e_{12}, e_{19}, and e_{21} are considered relevant, and we include events e_1 and e_5, as e_2 and e_6 are data dependent on the writes of local variable `done`. Exceptionally, a branch event can be considered irrelevant if that event is executed by a statement s enclosed within **`if(true*)`** in the program's NDSeq specification and all the events generated by that execution of s are irrelevant. Intuitively, this means that in the corresponding execution of the NDSeq specification, that particular execution of s can be entirely ignored without affecting the final outcome of the execution (by considering that

the nondeterministic **if(*true*∗)** will be resolved to **if**(**false**) in the corresponding NDSeq execution). Therefore, when inferring the NDSeq specification, we need to look for statements to add **if(*true*∗)** so that we can ignore events that are involved in conflict cycles. In the presence of the data dependencies between events, this becomes a combinatorial search problem.

In order to show that the execution Figure 7.3 is serializable, we need to ignore the conflict cycles formed by the thick solid arrows in the figure. For this, possible candidate events to ignore are: (1) the read e_3, (2) the write e_9, or (3) all three of e_{13}, e_{16}, and e_{18}. But, since the events e_9, e_{16}, and e_{18} affect the computation of **x**, they are relevant for the final result of the trace and they could not be eliminated in a matching serialization even if they were guarded by **if(*true*∗)**. Thus, our inference algorithm must focus on placing **if(*true*∗)** around events e_3 and e_{13}.

If we enclose an **if(*true*∗)** around Lines 5–10 as shown in Figure 7.2, we can safely mark event e_{14} irrelevant, because the branch e_{14} corresponds to is not evaluated, and thus, does not affect the rest of the execution. This also makes the events e_3, e_4, and e_{13} irrelevant because these events flow into only each other and e_{14}. Therefore, we can ignore the events e_3, e_4, e_{13}, and e_{14} together with the conflict cycles they are involved in. In fact, the only conflict cycles in the execution are formed by the events e_3 and e_{13}, and after ignoring these cycles, we can declare the execution in Figure 7.3 serializable. Serializing this execution respecting the remaining conflict edges gives us the execution trace in Figure 7.4. This trace can also be generated by a nondeterministic execution of the NDSeq specification given in Figure 7.2 by choosing *false* for **if(*true*∗)** in the first iteration of **i=1**.

MinCostSAT Solving for if(*true*∗) Placements. Having explained how a given parallel execution trace is checked against an existing NDSeq specification, we next explain how to infer such a specification. For this, we observe a set of representative parallel execution traces for which the standard conflict serializability check gives conflict cycles. Since we are inferring an NDSeq specification for the program, not for a single trace, using multiple traces allows us to observe variations in the executions and improves the reliability of the inferred NDSeq specification.

We then construct and solve a MinCostSAT formula that takes as input the events in the input traces and the conflict cycles detected by the standard conflict serializability check. While generating the formula, we encode the reasoning about relevant events and conflict cycles described above as constraints in the formula. In particular, the constraints enforce the data dependencies between the events and conditions to ignore all observed conflict cycles in the input traces. The MinCostSAT formulation contains variables corresponding to possible placement of **if(*true*∗)**s in the program, and the sum of these variables is minimized, so that as few **if(*true*∗)** as possible are added. If this formula is satisfiable, then the solution gives us a minimal set of statements $\mathcal{S}*$ in the program, such that the input traces are all serializable with respect to the NDSeq specification obtained by enclosing all statements in $\mathcal{S}*$ with **if(*true*∗)**. In other words, if there exists NDSeq specifications using which the

conflict cycles in the given traces can be safely ignored, our formulation gives us one of those specifications with the minimum number of nondeterministic branches.

Constraints are added to impose a number of conditions:

1. For each cycle of transactional conflicts, at least one of the events involved in the cycle must be made irrelevant. For example, we would add constraint $(X_{e_3} \vee X_{e_9} \vee X_{e_{13}})$ for the cycle between the `i=1` and `i=2` iterations by conflicts $e_3 \mapsto e_9$ and $e_9 \mapsto e_{13}$. This constraint enforces that at least one of the variables X_{e_3}, X_{e_9}, and $X_{e_{13}}$ be 1 in the solution.

2. Each event e can be made irrelevant only if all events that are data or control dependent on e are also irrelevant. For example, e_3 can be made irrelevant only if e_4, e_{13}, and e_{14} are made irrelevant, as well. For example, $(X_{e_3} \implies X_{e_4})$ is among the constraints added to model this requirement. The constraint enforces that whenever X_{e_3} is 1 in the solution, X_{e_4} be also 1 in the same solution.

3. For each event e, we add a constraint indicating that e is made irrelevant only if some **`if(true*)`** is added such that both: (a) some dynamic instance of the **`if(true*)`** contains e, and (b) no event contained by that dynamic instance is relevant.

   For example, an **`if(true*)`** around Line 5, Lines 5–7, or Lines 5–10 would make e_3 irrelevant, because none of events e_4, e_{13}, or e_{14} (which depend on e_3) are relevant. But an **`if(true*)`** around the entire **`while`** statement would not, because the dynamic **`if(true*)`** containing e_3 would also contain the relevant event e_{18}.

4. Finally, we forbid adding overlapping **`if(true*)`** constructs. For example, we forbid adding both an **`if(true*)`** around Lines 5 and 6 and one around Lines 6 and 7, as this would not be a well-structured program.

These constraints allow any solution that covers all of Lines 5–10, and no more, with some number of **`if(true*)`** constructs. (Because events e_3, e_4, e_{13}, and e_{14} all must be made irrelevant, and any larger **`if(true*)`** including these events would include relevant events). The minimal such solution places a single **`if(true*)`** that encloses Lines 5–10. Thus, our algorithm produces the correct NDSeq specification for this example.

7.2 Background: NDSeq Specifications

In this section, we recap some of the formal details from Chapter 6 of both NDSeq specifications and of our runtime algorithm for checking that a parallel execution conforms to its NDSeq specification. For this, we will assume that we are given an NDSeq specification a priori. We show in the next section how to infer such a specification with minimal nondeterminism.

$$g \in \mathit{Global} \quad l \in \mathit{Local} \quad x \in \mathit{Var} = \mathit{Global} \cup \mathit{Local}$$

$$\begin{aligned} s \in \mathit{Stmt} ::= {} & l = l \ \mathit{op}\ l \mid l = \text{constant} \mid l = l \mid g = l \mid l = g \mid s; s \\ & \mid \texttt{if}(l)\ s\ \texttt{else}\ s \mid \texttt{while}(l)\ s \mid \texttt{for}\ (l\ \texttt{in}\ l)\ s \\ & \mid \boxed{\texttt{coforeach}}\ (l\ \texttt{in}\ l)\ s \mid \boxed{\texttt{cobegin}}\ s; ...; s \\ & \mid \boxed{\texttt{atomic}}\ s \mid \boxed{\texttt{if(}\textit{true}*\texttt{)}}\ s \end{aligned}$$

Figure 7.5: Selected statements of our language. The constructs with a different semantics in the parallel program and the NDSeq specification are shown in gray color.

NDSeq Specifications and Parallelism Correctness

Given a parallel program $\mathcal{P}$, we specify the correct behavior of a parallel program by writing an equivalent nondeterministic sequential (NDSeq) program as a specification of $\mathcal{P}$, instead of explicitly giving a specification of the required input-output behavior. Informally, the equivalence means that for any input and thread schedule of $\mathcal{P}$ there exists an execution of the NDSeq program that produces the same output. We formalize these concepts next after describing the language constructs that we use to write parallel programs and their specifications.

We embed an NDSeq specification in the parallel program itself. The syntax for our language is shown in Figure 7.5. To simplify the presentation we consider a NDSeq program $\mathcal{P}$ to consist of a single procedure, with each global variable (in *Global*) referring to a distinct location on the shared heap and each local variable (in *Local*) referring to a distinct stack location of a thread.

Given a parallel program $\mathcal{P}$, the NDSeq specification is obtained by (a) overloading the parallel constructs that create threads (**coforeach** and **cobegin**) in a sequential context, and (b) introducing nondeterministic control flow with **if(*true*∗)**. Specifically, given an statement s in the parallel program, the user can modify the statement to **if(*true*∗)**$\{s\}$ in the NDSeq specification. For each program $\mathcal{P}$, given a set of statements to enclose with **if(*true*∗)**, we define two sets of executions *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$), described below. The correctness of the parallel program is then given by relating *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$).

Parallel Executions. *ParExecs*($\mathcal{P}$) contains the parallel executions of $\mathcal{P}$ where each **cobegin** and **coforeach** statement spawns new threads to execute its body. **cobegin** $s_1; ...; s_n$ is evaluated by executing each of $s_1, ..., s_n$ on a separate, newly created thread. **coforeach** is evaluated by executing each iteration of the loop on a separate, newly created thread.

NDSeq Executions. *NdSeqExecs*($\mathcal{P}$) contains the (nondeterministic) sequential executions of $\mathcal{P}$ where all statements are evaluated by a single thread. Under the sequential semantics, the statements other than **cobegin** and **coforeach** are interpreted in the standard

way. Statement **atomic** s is simply equivalent to s. Each evaluation of **cobegin** $s_1; ...; s_n$ is equivalent to running a nondeterministic permutation of statements $s_1, ..., s_n$. A statement **coforeach** is evaluated similarly to its deterministic version (**for**) except that the elements of the collection being iterated over are processed in a nondeterministic order. This, in essence, abstracts the semantics of the collection to an unordered set. Finally, **if(*true*∗)** indicates a nondeterministic branch. That is, each time a statement **if(*true*∗)**$\{s\}$ is evaluated, a boolean value is chosen for *true*∗ nondeterministically.

Parallelism Correctness. The parallelism correctness for $\mathcal{P}$ means that every final state reachable by a parallel execution of the program from a given initial state is also reachable by a NDSeq execution from the same initial state. Therefore, parallel executions have no unintended nondeterminism caused by thread interleavings: either the nondeterminism is prevented using synchronization, or it is expressed by the nondeterministic control flow in the sequential specification.

While defining correctness, we distinguish a set of global variables as *focus* variables, which are considered to be effective on the functionality of the program. Then, we reason about the equivalence executions by referring to the final valuation of the focus variables. For example, consider a parallel search algorithm. The variable pointing to the best (optimal) solution found is a focus variable, while statistics counters, which do not affect the final outcome of the search, are non-focus variables.

Definition 17 (Parallelism correctness)**.** *A program $\mathcal{P}$* conforms to *its NDSeq specification with respect to a set Focus$\subseteq$ Global iff for every parallel execution $E \in$ ParExecs($\mathcal{P}$), there exists a nondeterministic sequential execution $E' \in$NdSeqExecs($\mathcal{P}$), such that the initial states of E and E' are the same and the final states agree on the values of all variables in Focus.*

Runtime Checking of Parallel Executions

Recall that our our dynamic algorithm from Chapter 6, for checking whether or not an execution of parallel program conforms to its NDSeq specification, consists of two components: (1) a dynamic dataflow analysis to determine which parts of the execution are *relevant* to the final program result, and (2) a conflict-serializability [114, 12, 62] check on the relevant parts of the execution. We briefly recap these components here.

Execution Traces. Our checking algorithm operates on parallel executions traces τ, which are sequences of *events*. An event $e : T$ is one of:

$$T \quad ::= \quad x = x' \;\mid\; branch(l) \;\mid\; branch(true*)$$

indicating an assignment (of a local expression to a local or global variable, or of a global variable to a local variable), a branch (i.e., **while**(l) or **if**(l) on a local variable l), or a nondeterministic branch (an **if(*true*∗)**).

For any event, *Thread*(e) denotes the thread that generated the event e and *Guard*(e) denotes the event of type *branch*(*true*$*$) that corresponds to the most recent invocation of the innermost **if(*true*$*$)** that encloses the statement generating e. For events outside any **if(*true*$*$)** this value is *nil*.

Dynamic Data Dependence. In order to compute relevant events, we define the dynamic data-dependence relation $\dashrightarrow_\tau$ (and its transitive closure $\dashrightarrow^*_\tau$) on the events of trace τ by[1]:

D1. **(Data Dependence).** For each local variable read $e_j : x = l$ or branch e_j : *branch*(l), we add a dependence ($e_i \dashrightarrow_\tau e_j$) on the last $e_i : l = x'$ that comes before e_j in τ.

This dependence represents an actual dataflow through local l from e_i to e_j in the trace. Both of these events are in the same thread (since they operate on the same local) and their order and dependence will be the same in any serialization of the trace. These dependence edges are shown as thin dashed arrows in Figure 7.3.

D2. **(Inter-Thread Dependence).** For each global variable read $e_j : l = g$ we add dependencies ($e_i \dashrightarrow_\tau e_j$) on events $e_i : g = l'$ as follows. From each thread we pick the last write to g that comes before e_j in τ, and the first write to g that comes after e_j in τ. This conservative dependence is necessary because the relative order of reads and writes to a global variable from different threads may change in a serialization of the trace. In this way, dependencies are preserved while reordering the accesses.

Relevant Events. In Chapter 6, we describe an algorithm (**ComputeRelevant** in Figure 6.8) for computing the set of relevant events given an execution trace τ of program and an NDSeq specification for the program — consisting of the set $\mathcal{S}*$ that are immediately enclosed with **if(*true*$*$)** and the set *Focus* of global variables containing the final result of the program's computation.

Here we define the set *Relevant*($\tau, \mathcal{S}*$, *Focus*) of relevant events computed by algorithm **ComputeRelevant**. Let E_s denote the set that contains exactly the events generated by a dynamic instance of statement s in trace τ. Let *NdBlock*(e) return the smallest set E_s such that $e \in E_s$ and $s \in \mathcal{S}*$. In other words, *NdBlock*(e) gives the execution of the smallest statement that generated e and is enclosed with **if(*true*$*$)** in the NDSeq specification of the program. If e is not generated by a statement enclosed with **if(*true*$*$)**, then *NdBlock*(e) is undefined. Then:

Definition 23 (Relevant events)**.** *Define Relevant*($\tau, \mathcal{S}*$, *Focus*) *as the smallest set of events from trace* τ *such that:*

R1. If $e : g = l$ *is a write to global* $g \in$ *Focus and* e *is the last write to* g *in Thread*(e)*, then* $e \in$ *Relevant*($\tau, \mathcal{S}*$, *Focus*)*.*

[1]We omit the subscript when τ is clear from the context.

R2. If e : branch(l) is a branch and NdBlock(e) is undefined, then $e \in Relevant(\tau, \mathcal{S}*, Focus)$.

R3. If e : branch(l) is a branch, NdBlock(e) $= E_s$, and there is an event $e' \in E_s$ such that $e' \in Relevant(\tau, \mathcal{S}*, Focus)$, then $e \in Relevant(\tau, \mathcal{S}*, Focus)$.

R4. If exists an $e' \in Relevant(\tau, \mathcal{S}*, Focus)$ with $e \dashrightarrow^*_\tau e'$, then $e \in Relevant(\tau, \mathcal{S}*, Focus)$. (Note that, in this case e is an assignment event.)

R1 makes all final writes to focus variables relevant. R2 and R3 state the condition for a (deterministic) branch event e to become relevant: either if e is not part of any execution of a statement enclosed with **`if(true`**$*$**`)`** in the NDSeq specification, or if the smallest such execution generating e already contains another relevant event. R4 makes an event relevant if it flows into another relevant event. Notice that, a dynamic instance of a statement $s \in \mathcal{S}*$ becomes totally irrelevant, if it does not contain a final write to a focus variable and none of the events generated by that instance flow (through $\dashrightarrow$) into other relevant events outside the instance.

Conflict-Serializability Checking. In Chapter 6, we defined the conflict relation between individual events with respect to a set of events, denoted $\mathcal{E}$. For traditional conflict serializability checking $\mathcal{E}$ is instantiated as the set of all events in a trace.

Definition 18 (Conflicting events in a set of events $\mathcal{E}$)**.** *Given a set $\mathcal{E}$ of events from a trace τ, two events $e, e' \in \tau$ are conflicting in $\mathcal{E}$ (written $e \rightsquigarrow_{\mathcal{E}} e'$) iff (a) $e, e' \in \mathcal{E}$, and (b) e occurs before e' in τ, and (c) both events operate on the same shared global variable, and at least one of them represents a write, and (d) the events are generated by different threads.*

Further, we lifted the conflict relation from events to threads. When comparing two threads for conflicts we need to consider their events and all the events of their descendant threads. Thus, for a thread t we define its *transaction* as the set of events *Trans* (t) that includes all the events of t and of the descendant threads of t.

Definition 19 (Conflicting threads with respect to a set of events $\mathcal{E}$)**.** *Given a set $\mathcal{E}$ of events from a trace τ, two threads t, t' are conflicting in trace τ (written $t \rightsquigarrow_{\mathcal{E}} t'$) iff (a) their transaction sets are disjoint (i.e., one is not a descendant of the other), and (b) there exist two events $e \in$ Trans (t) and $e' \in$ Trans (t') that are conflicting ($e \rightsquigarrow_{\mathcal{E}} e'$). The relation $t \rightsquigarrow^*_{\mathcal{E}} t'$ is the transitive and reflexive closure of the thread conflict relation.*

We can then restate Theorem 22 from Chapter 6 as:

Theorem 24. *Let τ be a parallel execution of $\mathcal{P}$, with NDSeq specification ($\mathcal{S}*, Focus$). If the transactional conflict relation $\rightsquigarrow_{Relevant(\tau, \mathcal{S}*, Focus)}$ is acyclic on the threads of τ, then τ conforms to the NDSeq specification of $\mathcal{P}$. That is, there exists a nondeterministic sequential execution of $\mathcal{P}$'s NDSeq specification from the same initial state as τ and with a final state that agrees with τ on the values of all Focus variables.*

7.3 Inferring a Suitable NDSeq Specification

Having explained our runtime approach to reasoning about whether a program obeys its NDSeq specifications, as well as the role of **`if(true*)`** annotations, the key question is where should users add such **`if(true*)`** annotations? We propose an algorithm to automatically infer the correct placement of **`if(true*)`**. In particular, our goal is, given a set of traces $\mathcal{T}$ of $\mathcal{P}$, to come up with a set $\mathcal{S}*$ of statements so that if we immediately enclose the statements in $\mathcal{S}*$ with **`if(true*)`** and compute the relevant events in $\mathcal{T}$ (as in Definition 23), conflict serializability checking over these relevant events gives no serializability violations, showing that all the traces in $\mathcal{T}$ conform to the NDSeq specification obtained from $\mathcal{S}*$. We assume that the user still indicates which variables are in *Focus*.

In order to infer the NDSeq specification, we formulate the computation of relevant events as solving a Boolean Satisfiability (SAT) instance. In contrast to our original algorithm for computing *Relevant*$(\tau, \mathcal{S}*, \mathit{Focus})$ in Chapter 6, formulating our reasoning as a constraint solving problem allows us to not only check that a given trace conforms to an NDSeq specification. It also enables us to perform such checking without giving an NDSeq specification. Instead, the SAT solver finds a suitable set $\mathcal{S}*$ of statements to enclose with **`if(true*)`**, so that, with respect to the resulting relevant events, all traces in $\mathcal{T}$ are conflict serializable.

We also need to be careful with the set $\mathcal{S}*$, because **`if(true*)`**s could add extra behaviors to the NDSeq specification of the program and those extra behaviors should not violate the functional correctness of the program. Therefore, we need to find the minimum set $\mathcal{S}*$ to which all traces in $\mathcal{T}$ conform. For this, we then turn the problem into a Minimum-Cost Boolean Satisfiability Problem (MinCostSAT).

SAT Formulation for Inferring NDSeq Specification

We start with a program $\mathcal{P}$ where a set *Focus* of focus variables are marked by the programmer, but no statement is enclosed with **`if(true*)`** — i.e., the set $\mathcal{S}*$ is empty. We are given a set $\mathcal{T}$ of parallel execution traces, and our goal is to determine if there exist an NDSeq specification that encloses statements $\mathcal{S}*$ with **`if(true*)`**, with which we can show that all the conflict cycles can be ignored safely — i.e., each conflict cycle contains at least one irrelevant event, so all traces in $\mathcal{T}$ conform to the NDSeq specification. For this, we construct and solve a SAT instance on the following (boolean) indicator variables:

- X_s, for each statement s in $\mathcal{P}$.
- X_e, for each event e in any of the traces in $\mathcal{T}$. Note that these dynamic events are uniquely identified, both in a trace and across all traces.
- X_{E_s}, for each dynamic execution of a statement s generating exactly the events in E_s in any of the traces in $\mathcal{T}$.

Let $\mathbf{X}$ denote a solution to a SAT instance. We refer to the values (from the set $\{0, 1\}$) of indicator variables X_e, X_s, and X_{E_s} in solution $\mathbf{X}$ by $\mathbf{X}_e$, $\mathbf{X}_s$, and $\mathbf{X}_{E_s}$, respectively.

We will construct our constraints to guarantee that, if our instance has a solution $\mathbf{X}$, then there exists an NDSeq specification for $\mathcal{P}$ to which all traces in $\mathcal{T}$ *conform*, and the following hold:

1. The set $\mathcal{S}* = \{s \mid \mathbf{X}_s = 1\}$ contains the statements we need to surround with **`if(true*)`** in the inferred NDSeq specification.

2. For each event e in a trace $\tau \in \mathcal{T}$, if $\mathbf{X}_e = 1$ then event e is *irrelevant* in τ — i.e., $e \notin \mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ with respect to the inferred NDSeq specification.

3. For each X_{E_s}, if $\mathbf{X}_{E_s} = 1$, then $\mathbf{X}_s = 1$, and thus, statement s will be enclosed with an **`if(true*)`**, and that **`if(true*)`** around s will make events in E all irrelevant (i.e., $\mathbf{X}_e = 1$ for all $e \in E$).

4. For each conflict cycle C in some trace $\tau \in \mathcal{T}$, $\mathbf{X}_e = 1$ for at least one event e in C. This means, cycle C will not be observed when checking conflict serializability over $\mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ computed using the set $\mathcal{S}*$ given by solution $\mathbf{X}$.

We construct the full SAT instance as follows. The conditions R1 through R4 are from Definition 23 of relevant events.

(A) Condition R1 dictates that if an event e is a final write to a variable in *Focus*, then e is relevant. Thus, for each such event e, we add the constraint:

$$\neg X_e \tag{7.1}$$

For efficiency, we actually just substitute 0 for each such X_e.

(B) Conditions R2 and R3 dictate that a branch event e — i.e., of type $\mathit{branch}(l)$ for some local l– becomes irrelevant only if (a) e is generated by a dynamic instance of a statement s which is directly enclosed by an **`if(true*)`** and (b) all the events generated by that instance are irrelevant. To ensure (a) and (b) we add the following constraints.

- For each branch event e we add the constraint:

$$X_e \implies \bigvee_{e \in E_s} X_{E_s} \tag{7.2}$$

- For each dynamic instance of statement s, producing events E, we add constraint:

$$X_{E_s} \implies X_s \tag{7.3}$$

- For each dynamic instance of s, producing events E, and for each $e \in E$, we add:

$$X_{E_s} \implies X_e \tag{7.4}$$

Therefore, if a branch event e needs to be irrelevant, the solver must find a dynamic instance of some s where all of its events (including e) are irrelevant, and s must be enclosed with an **`if(true*)`**. Together these constraints ensure that some dynamic *branch*(*true**) event, corresponding to an **`if(true*)`** directly enclosing statement s, can be made irrelevant only if an **`if(true*)`** has been added around statement s (that is, $\mathbf{X}_s = 1$) and if every event e guarded by the *branch*(*true**) is irrelevant.

(C) Condition R4 dictates that if an event e' is relevant and if there is another event e such that $e \dashrightarrow^*_\tau e'$, then e must also be relevant. In other words, if e needs to be irrelevant, e' must be irrelevant, too. To ensure this, we add the following constraint for each pair of events e, e' such that $e \dashrightarrow^*_\tau e'$ for some τ:

$$X_e \implies X_{e'} \tag{7.5}$$

(D) Given two statements s and s', we say that s *overlaps* with s' if s is not nested inside s', s' is not nested inside s, and there is a statement s'' nested inside both s and s'.

If we have two overlapping statements, then we cannot surround both of them by **`if(true*)`** because such an action would result in an invalid program. For example, consider the sequential composition of three statements $s1; s2; s3$. Then statements $s1; s2$ and $s2; s3$ overlap with each other and it is easy to see that we cannot surround both of them with **`if(true*)`** simultaneously. Our constraint system ensures this restriction by adding, for every pair of overlapping statements s, s':

$$X_s \implies \neg X_{s'} \tag{7.6}$$

(E) Finally, we need to ensure that $\rightsquigarrow_{Relevant(\tau,\mathcal{S}*,Focus)}$ is acyclic on the threads of τ. We encode in our SAT instance the check for transactional conflict cycles – that is:

- For all threads t, t' in τ, introduce variables $X_{t,t'}$ to indicate whether or not $t \rightsquigarrow_{Relevant(\tau,\mathcal{S}*,Focus)} t'$.
- For each triple of distinct t, t', and t'', we add the constraint:

$$X_{t,t'} \wedge X_{t',t''} \implies X_{t,t''} \tag{7.7}$$

- For each pair of distinct threads t and t' for which *Trans* (t) and *Trans* (t') are disjoint, and for each pair $e \in$ *Trans* (t) and $e' \in$ *Trans* (t') for which $e \rightsquigarrow e'$, we add the constraint:

$$\neg X_e \wedge \neg X_{e'} \implies X_{t,t'} \tag{7.8}$$

  That is, if e and e' are both relevant, then $t \rightsquigarrow_{Relevant(\tau,\mathcal{S}*,Focus)} t'$.
- For each threat t, we add the constraint that there can be no cycles from t back to t:

$$\neg X_{t,t} \tag{7.9}$$

We prove in Section 7.4 that any solution to the above constraints gives us an NDSeq specification to which all the traces in $\mathcal{T}$ conform. In particular, if $\mathbf{X}$ is a solution to the constraint system, then $\mathcal{S}* = \{s \mid \mathbf{X}_s = 1\}$ is the set of statements to enclose with **if(*true*∗)** in the NDSeq specification.

A noteworthy implication of this fact is that, if there is no way to show that a trace in $\mathcal{T}$ is serializable by adding **if(*true*∗)** around program statements, then the solver must report that the constraints are unsatisfiable. Thus, an unsatisfiable instance indicates that one of the traces in $\mathcal{T}$ is likely to contain parallelism errors, such as atomicity violations.

MinCostSAT Solving for a Minimal NDSeq Specification

The SAT formulation above guarantees that if there is an NDSeq specification, in which a set $\mathcal{S}*$ of statements are enclosed in **if(*true*∗)**, to which the traces in $\mathcal{T}$ conform, then there exists a solution $\mathbf{X}$ that selects $\mathcal{S}*$ — i.e., forall $s \in \mathcal{S}*$, $\mathbf{X}_s = 1$. But, we have not guaranteed that such a solution selects exactly $\mathcal{S}*$. In other words, the solution may tell us to enclose with **if(*true*∗)** more statements than those in $\mathcal{S}*$. In this case, there is a risk that some statements can be unnecessarily surrounded by **if(*true*∗)**. Since adding **if(*true*∗)** may cause adding more behaviors to the program, one could end up adding **if(*true*∗)**s that violate functional correctness. Therefore, we need a mechanism to find a solution to the constraint system so that functional correctness is not broken. We noticed that if we add a minimal number of **if(*true*∗)**s then there is a lower chance of breaking the functional correctness. For this, we re-formulate the SAT problem above as a MinCostSAT problem.

MinCostSAT is a special form of SAT, where, in addition to the constrains above, a cost function $\mathbf{C}$ assigns each variable a non-negative cost. The solver is asked to find a solution that not only satisfies all the constraints, but also minimizes the sum of the costs of the variables that are assigned 1 in the solution.

Our MinCostSAT formulation contains all the constraints (A)–(E) given above. In addition, we define the cost function $\mathbf{C}$ such that for each X_s, $\mathbf{C}(X_s) = 1$, and for other variables $X_\bullet$, $\mathbf{C}(X_\bullet) = 0$. Therefore, the solver optimizes the objective[2]:

$$minimize \quad \sum_{s \ in \ \mathcal{P}} \mathbf{X}_s \tag{7.10}$$

In this formulation, X_s is assigned 1 only when a branch event e must be marked irrelevant to discharge a conflict cycle, and for this, s must be surrounded with **if(*true*∗)**. Otherwise, X_s is assigned 0 to minimize the cost.

Note that, adding only the minimum number of necessary **if(*true*∗)**s to the inferred NDSeq specification is a heuristic to reduce the risk of violating the functional specification. In other words, if we find a solution to our MinCostSAT formulation above, then we have

[2]This formulation can also be mapped to a Partial Maximum Satisfiability problem (PMAX-SAT), which contains our constraints in (A)–(E) as hard constraints and for each variable X_s a soft constraint ($\neg X_s$); the objective is to satisfy all hard constraints and maximum number of soft constraints.

inferred a *likely* NDSeq specification of the parallel program, and that specification may still violate the functional correctness specification of the program. If we find no solution, then probably there is no NDSeq specification for the parallel program. Thus, we foresee a repetitive process for finding the right NDSeq specification, in which the user sequentially checks the functional correctness specification (e.g., assertions) after inferring an NDSeq specification, and if any functional correctness criterion is violated, rules out the current placement of **`if(true*)`**s for the next iteration of NDSeq specification inference.

Optimizations

We conclude this section by presenting two optimizations that we observed to have significant effect in simplifying the constraint system, and thus reducing the MinCostSAT solving time from minutes to several seconds.

Using Dynamic Slicing. Recall that in Constraint 7.1 we pre-assign 0 to each X_e if e is a final write to a focus variable; the values for indicator variables of other events are computed during the SAT solving. By using the dynamic slice [5] of the trace, we can improve this by providing values for more variables before the SAT solving.

Let $\hookrightarrow_\tau$ denote the control dependence between the events in trace τ, and $\longrightarrow_\tau^*$ denote the transitive closure of $(\dashrightarrow_\tau \cup \hookrightarrow_\tau)$. Recall that $\dashrightarrow_\tau$ denotes data dependence defined in Section 7.2. (See Section 7.5 for the formal definition of $\hookrightarrow_\tau$.)

A *dynamic slice* of a trace τ with respect to the focus variables, denoted by $DSlice(\tau, Focus)$, is the set of events from τ such that $e \in DSlice(\tau, Focus)$ *iff* there exists an event $e' : g = l$ in τ, e' is the last write to $g \in Focus$ in $Thread(e')$, and $e \longrightarrow_\tau^* e'$.

We prove in Section 7.5 that:

$$\forall \mathcal{S}*.\ DSlice(\tau, Focus) \subseteq Relevant(\tau, \mathcal{S}*, Focus)$$

That is, given an input trace $\tau \in \mathcal{T}$ to our MinCostSAT formulation, the set of relevant events in τ given by a solution will always be a superset of the dynamic slice of τ; this result holds for any inferred NDSeq specification. In other words, if $e \in DSlice(\tau, Focus)$ then e must be relevant. Thus, we can safely modify (A), which pre-assigns 0 to X_e only if e is a final write to a focus variable, in Section 7.3 as follows.

(A') For each event $e \in DSlice(\tau, Focus)$, we add the constraint:

$$\neg X_e \tag{7.1'}$$

Thus, we substitute 0 for each such e in the constraint system.

Grouping Events. The formulation of MinCostSAT in Section 7.3 considers all the events in the execution and the dependencies between those events. This could lead to large MinCostSAT instances that are expensive to solve. We address this situation by grouping events

into disjoint sets. Whenever we see an execution of a statement that is completely excluded from the dynamic slice, we treat all the events $\mathbf{e} = \{e_1, ..., e_n\}$ in that dynamic execution instance of the statement as a single (compound) event. For each $i \in [1, n]$, we then replace the variable X_{e_i} in all constraints of the MinCostSAT by the variable $X_{\mathbf{e}}$ and we lift the constraints described in Section 7.3 to sets of events. In this way, we can ignore the dependency relationship between the events within a group and concentrate on inter-group dependencies. Although grouping events in this way may result in less optimal solutions (with higher cost than the original and more general formulation in Section 7.3), in our experiments we confirmed that this optimization did not affect the inferred specification for our benchmarks.

7.4 Correctness of Specification Inference Algorithm

Let $\mathcal{T}$ be a set of parallel execution traces of program $\mathcal{P}$, and *Focus* be a set of focus variables. Let $\mathbf{X}$ be a satisfying solution to the SAT formulation in Section 7.3, and $\mathcal{S}*$ contain the set of statements to enclose with **`if(true*)`** in the inferred NDSeq specification — i.e., $s \in \mathcal{S}*$ *iff* $\mathbf{X}_s = 1$.

Note that we have already proved in Theorem 22, in Chapter 6, that it is sound to check conflict serializability for each trace τ in $\mathcal{T}$ only on the events in $Relevant(\tau, \mathcal{S}*, Focus)$. That is, if we find no conflict cycles after omitting irrelevant events, then each τ conforms to the inferred NDSeq specification.

To prove the soundness of our inference algorithm, we first prove the soundness of its embedded relevance calculation:

Lemma 25. *Let $\mathcal{T}$ be a set of parallel execution traces of program $\mathcal{P}$. Let $\mathcal{S}*$ be a set of* **`if(true*)`** *inferred by the above algorithm, given $\mathcal{T}$ and focus variables Focus – that is, $\mathcal{S}*$ corresponds to a solution $\mathbf{X}$ of the constraint system built by the inference algorithm.*

For each event e in $\tau \in \mathcal{T}$, if $e \in Relevant(\tau, \mathcal{S}*, Focus)$, then $\mathbf{X}_e = 0$.

Proof. Recall the conditions R1–R4 in Definition 23 for an event $e \in \tau$ to be in the set $Relevant(\tau, \mathcal{S}*, Focus)$. Given these conditions, think of an iterative procedure to compute the relevant events: $Relevant(\tau, \mathcal{S}*, Focus)$ is initialized to an empty set, and at each step, one of the rules R1-R4 is applied to add a new event to $Relevant(\tau, \mathcal{S}*, Focus)$, until $Relevant(\tau, \mathcal{S}*, Focus)$ does not change. We do the proof by induction on the length of this iteration. The base case where $Relevant(\tau, \mathcal{S}*, Focus) = \emptyset$ is trivial. In the following, we do a case split on the condition that causes event e to be added to set $Relevant(\tau, \mathcal{S}*, Focus)$.

R1 In this case, e is the last write to a focus variable by one of the threads. Therefore, by Constraint 7.1, $\mathbf{X}_e$ is always set to 0.

R2 In this case, e is a $branch(l)$ event and $NdBlock(e)$ is undefined — i.e., e is not generated by a statement enclosed with **`if(true*)`**. Consider each X_{E_s} term in Constraint 7.2:

$$X_e \implies \bigvee_{e \in E_s} X_{E_s}$$

For each dynamic execution E_s such that $e \in E_s$, s is not enclosed with **`if(true*)`** — otherwise *NdBlock*(e) would be defined. Thus, $s \notin \mathcal{S}*$ so $\mathbf{X}_s = 0$. Therefore, by Constraint 7.3 each such $\mathbf{X}_{E_s}$ is 0. Therefore, by Constraint 7.2, we have $\mathbf{X}_e = 0$.

R3 In this case, e is a *branch*(l) event, *NdBlock*$(e) = E_s$, and there is an event $e' \in E_s$ such that $e' \in$ *Relevant*$(\tau, \mathcal{S}*,$ *Focus*$)$. By the inductive hypothesis, $\mathbf{X}_{e'} = 0$.

Let $E'_{s'}$ contain e — that is, $X_{E'_{s'}}$ appears in the disjunction in Constraint 7.2 for e:

$$X_e \implies \bigvee_{e \in E'_{s'}} X_{E'_{s'}}$$

We will show by case analysis that $\mathbf{X}_{E'_{s'}} = 0$ for any such $E'_{s'}$. Thus, $\mathbf{X}_e = 0$.

Suppose $e' \in E'_{s'}$, then $\mathbf{X}_{E'_{s'}} = 0$ by Constraint 7.4.

Suppose instead that $e' \notin E'_{s'}$, and that $E'_{s'} \subseteq E_s$. Then, because $e \in E'_{s'}$ and because *NdBlock*$(e) = E_s$ is the *minimal* E_s both containing e and for which $\mathbf{X}_s = 1$, it must be the case that $\mathbf{X}_{s'} = 0$. Thus, by Constraint 7.3, we have $\mathbf{X}_{E'_{s'}} = 0$.

Suppose instead that $e' \notin E'_{s'}$ and that $E'_{s'} \not\subseteq E_s$. Then, statements s and s' must be *overlapping* (i.e., they contain a common statement, but neither statement contains the other), and as $\mathbf{X}_s = 1$, by Constraint 7.6 we have $\mathbf{X}_{s'} = 0$. Thus, by Constraint 7.3, we have $\mathbf{X}_{E'_{s'}} = 0$.

R4 In this case, $e \dashrightarrow^* e'$ for some e' already in *Relevant*$(\tau, \mathcal{S}*,$ *Focus*$)$. Therefore, by inductive hypothesis, $\mathbf{X}_{e'} = 0$. Then, $\mathbf{X}_e = 0$ by Constraint 7.5. □

Theorem 26 (Soundness of Inference)**.** *Given a set $\mathcal{T}$ of parallel execution traces of program $\mathcal{P}$ and focus variables Focus, suppose that our SAT instance is satisfiable and let $\mathcal{S}*$ be the set of statements, inferred from the solution, to be enclosed with* **`if(true*)`***. Then, every trace in $\mathcal{T}$ conforms to the NDSeq specification given by Focus and $\mathcal{S}*$.*

Proof. Suppose some $\tau \in \mathcal{T}$ does not conform to the inferred NDSeq specification. That is, there exist events $e_1, \ldots, e_k$ from τ that are all relevant and that form a cycle of conflicts between the threads of τ. (Note that the $e_1, \ldots, e_k$ are all events of type "$x = x'$", and all read from or write to a global variable.)

The inferred **`if(true*)`** locations $\mathcal{S}*$ correspond to a solution $\mathbf{X}$ to the constraint system built by our inference algorithm. By Lemma 25, because $e_1, \ldots, e_k$ are all relevant:

$$\mathbf{X}_{e_1} = \mathbf{X}_{e_2} = \cdots = \mathbf{X}_{e_k} = 0$$

Because $e_1, \ldots, e_k$ form a cycle of conflicts, we have $e_1 \rightsquigarrow e_2$, $e_2 \rightsquigarrow e_3$, …, $e_k \rightsquigarrow e_1$. Thus, as all $\mathbf{X}_{e_i} = 0$, by Constraint 7.8, we have:

$$\mathbf{X}_{Thread(e_1),Thread(e_2)} = \mathbf{X}_{Thread(e_2),Thread(e_3)} = \cdots = \mathbf{X}_{Thread(e_k),Thread(e_1)} = 1$$

Thus, by repeated application of Constraint 7.7, we have $\mathbf{X}_{Thread(e_1),Thread(e_1)} = 1$. But this contradicts Constraint 7.9, which requires $\neg\mathbf{X}_{Thread(e_1),Thread(e_1)}$.

Thus, all $\tau \in \mathcal{T}$ must conform to the inferred NDSeq specification. □

We similarly prove that our inferrence algorithm is *complete* with respect to our runtime checking algorithm from Chapter 6.

Theorem 27 (Completeness of Inference)**.** *Given a set $\mathcal{T}$ of parallel execution traces of program $\mathcal{P}$ with focus variables Focus, suppose that there exists some NDSeq specification $\mathcal{S}*$, Focus to which every trace τ in $\mathcal{T}$ conforms. Then, there exists a solution to the SAT instance generated by our inference algorithm, and thus the inference algorithm will infer a NDSeq specification for $\mathcal{P}$, $\mathcal{T}$, and Focus.*

Proof. Consider the following valuation of $\mathbf{X}$:

- $\mathbf{X}_s = 1$ iff $s \in \mathcal{S}*$
- $\mathbf{X}_e = 0$ iff $e \in \mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ for some $\tau \in \mathcal{T}$
- $\mathbf{X}_{E_s} = 1$ iff $s \in \mathcal{S}*$ and $e \notin \mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ for all $e \in E_s$ and $\tau \in \mathcal{T}$
- $\mathbf{X}_{t,t'} = 1$ iff $t \rightsquigarrow^*_{Relevant(\tau,\mathcal{S}*,Focus)} t'$ for some $\tau \in \mathcal{T}$

We will show that $\mathbf{X}$ is a solution to the SAT instance generated by our specification inference algorithm – that is, $\mathbf{X}$ satisfies all of Constraints (7.1)–(7.9).

(7.1) Let e is the final write in its thread to a *Focus* variable. Then $e \in \mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ by condition (R1) in the definition (Definition 23) of relevant events. Thus, $\mathbf{X}_e = 0$ by construction, satisfying Constraint 7.1.

(7.2) Let e be a $\mathit{branch}(l)$ event. If e is relevant, then $\mathbf{X}_e = 0$ by construction and Constraint 7.2 is trivially satisfied. Suppose instead that e is not relevant, and thus $\mathbf{X}_e = 1$ by construction. Then, there must exist some E_s such that $\mathit{NdBlock}(e) = E_s$, or else e would be relevant by (R2) of Definition 23. Thus, $s \in \mathcal{S}*$. Further, there can be no relevant e' in E_s, or else e would be relevant by condition (R3). Thus, $\mathbf{X}_{E_s} = 1$ by construction, satisfying Constraint 7.2.

(7.3) Let E_s be the events produced by some dynamic instance of statement s. If $\mathbf{X}_s = 1$, then Constraint 7.3 is trivially satisfied. Suppose instead that $\mathbf{X}_s = 0$ — that is, $s \notin \mathcal{S}*$. Then, $\mathbf{X}_{E_s} = 0$ by construction, satisfying Constraint 7.3.

(7.4) Let $e \in E_s$. If $\mathbf{X}_e = 1$, then Constraint 7.4 is trivially satisfied. Suppose instead that $\mathbf{X}_e = 0$ — that is, that e is relevant. Then, by construction $\mathbf{X}_{E_s} = 0$, so Constraint 7.4 is satisfied.

(7.5) Let $e \dashrightarrow^*_\tau e'$ for some $\tau \in \mathcal{T}$. If $\mathbf{X}_{e'} = 1$, then Constraint 7.5 is trivially satisfied. Suppose instead that $\mathbf{X}_{e'} = 0$ — that is, e' is relevant. Then, by (R4) of Definition 23, event e is also relevant. Thus, $\mathbf{X}_e = 0$ by construction, satisfying Constraint 7.5.

(7.6) If s and s' are overlapping statements, then it cannot be the case that both s and s' are in $\mathcal{S}*$, as $\mathcal{S}*$ corresponds to a valid NDSeq specification. Thus, by construction $\mathbf{X}_s = 0$ or $\mathbf{X}_{s'} = 0$, satisfying Constraint 7.6.

(7.7) Relation $\rightsquigarrow^*_{Relevant(\tau,\mathcal{S}*,Focus)}$ is transitive, so Constraint 7.7 is always satisfied.

(7.8) Let t and t' be two threads from some $\tau \in \mathcal{T}$. If $\mathbf{X}_{t,t'} = 1$, then Constraint 7.8 is trivially satisfied. Suppose instead that $\mathbf{X}_{t,t'} = 0$ — that is, it is *not* the case that $t \rightsquigarrow^*_{Relevant(\tau,\mathcal{S}*,Focus)} t'$. Then, by the definition of the transactional conflict relation (Definition 19), for any $e \in Trans\ (t)$ and $e' \in Trans\ (t')$, either e or e' is irrelevant. Thus, by construction $\mathbf{X}_e = 1$ or $\mathbf{X}_{e'} = 1$, satisfying Constraint 7.8.

(7.9) Each trace $\tau \in \mathcal{T}$ conforms to the NDSeq specification *Focus*,$\mathcal{S}*$, so for no t is it the case that $t \rightsquigarrow^*_{Relevant(\tau,\mathcal{S}*,Focus)} t$. Thus, Constraint 7.9 is always satisfied. □

7.5 Correctness of Dynamic Slicing Optimization

To introduce the dynamic slicing, we need to define the control dependence relation $\hookrightarrow_\tau$ between events of a trace τ.

Definition 28. *Let e_i and e_j be events from trace τ. Event e_j is* control dependent *on e_i, denoted $e_i \hookrightarrow_\tau e_j$, iff:*

1. *e_i is a branch(l) event generated by the execution of a* **`if`**(l), **`while`**(l), *or* **`for`** *statement,*
2. *e_j is generated by the execution of a statement s' nested inside the above conditional statement s statement, and*
3. *the statement s' generating e_j is not nested inside any other conditional/loop statements that are themselves nested inside the above statement.*

Let $\longrightarrow^*_\tau$ denote the transitive closure of $(\dashrightarrow_\tau \cup \hookrightarrow_\tau)$.

Note that our definition of control dependence is particularly simple, because we describe our dynamic slicing optimization on a language with no unstructured control flow (i.e., **`break`** or **`continue`**). As with our NDSeq runtime checking algorithm and NDSeq specification inference algorithm, control dependence and our dynamic slicing algorithm can be easily adapted to standard unstructured control flow.

Definition 29. *A* dynamic slice *of a trace with respect to focus variables Focus is defined as follows. Let Target*$(\tau, Focus)$ *denote the set of all evennts that directly affect the final output:*

$$Target(\tau, Focus) \;=\; \{e : g = l \in \tau \mid g \in Focus \wedge e \text{ is last write to } g \text{ in Thread } (e)\}$$

Then a dynamic slice of a trace τ*, denoted by DSlice*$(\tau, Focus)$*, is the set:*

$$DSlice(\tau, Focus) \;=\; \{e \in \tau \;\mid\; \exists e' \in Target(\tau, Focus) \text{ such that } e \longrightarrow_\tau^* e'\}$$

Lemma 31 below, enables us to improve the efficiency of solving our MinCostSAT instances by performing the optimization given in Section 7.3 — that is, setting $X_e = 0$ for any event e in $DSlice(\tau, Focus)$. Further, Lemma 30 allows us to modify Constraint 7.5 in Section 7.3 by replacing $\dashrightarrow_\tau^*$ with $\longrightarrow_\tau^*$:

(C') Condition R4 and Lemma 30 dictate that if an event e' is relevant and if there is another event e such that $e \longrightarrow_\tau^* e'$, then e must also be relevant. In other words, if e needs to be irrelevant, e' must be irrelevant, too. To ensure this, we add the following constraint for each pair of events e, e' such that $e \longrightarrow_\tau^* e'$ for some τ:

$$X_e \implies X_{e'} \tag{7.5'}$$

Lemma 30. *Let* τ *be a parallel execution trace of program* $\mathcal{P}$*, with NDSeq specification* $\mathcal{S}*$*, Focus, and let e be a relevant event in* τ *— i.e.,* $e \in Relevant(\tau, \mathcal{S}*, Focus)$*. Then, for any* e' *such that* $e' \longrightarrow_\tau^* e$*, event* e' *must also be relevant.*

Proof. We prove by induction that, for any e' such that $e' \longrightarrow_\tau^* e$, event e' is relevant.

The base case, in which $e' = e$, trivially holds.

Suppose $e' \longrightarrow_\tau e''$ for some e'' such that $e'' \longrightarrow_\tau^* e$ and e'' is relevant. We will prove by case analysis that e' must also be relevant.

- Suppose that $e' \dashrightarrow_\tau e''$. Then, by (R4) in Definition 23, event e' is also relevant.
- Suppose instead that $e' \hookrightarrow_\tau e''$. In this case, e' must be a $branch(l)$ event generated by some conditional or loop statement, and e'' must be generated by some statement in the body of that conditional/loop.

  Suppose that $NdBlock(e')$ is undefined. By (R2) in Definition 23, event e' is relevant.

  Suppose instead that $NdBlock(e') = E_s$. Then, we must have that $e'' \in E_s$, because event $e' \in E_s$ and e'' is generated by a statement in the body of the conditional/loop statement that generated e'. Thus, by (R3) in Definition 23, event e' is relevant. □

Lemma 31. *Let* τ *be a parallel execution trace of program* $\mathcal{P}$*, with NDSeq specification* $\mathcal{S}*$*, Focus. Then,* $DSlice(\tau, Focus) \subseteq Relevant(\tau, \mathcal{S}*, Focus)$*.*

Proof. Suppose event $e \in DSlice(\tau, Focus)$. Then, there exists some event $e' \in Target(\tau, Focus)$ such that $e \longrightarrow_\tau^* e'$.

By (R1) in Definition 23, event e' is relevant because e' is the last write in its Thread to some *Focus* variable. Thus, by Lemma 30, event e is relevant. □

Note that the above two optimizations cannot affect the *soundness* of the inference algorithm. Both only add constraints to our MinCostSAT instance (and simplify the instance by substituting $X_e = 0$ for some e for which the constraint $\neg X_e = 0$ is added). Thus, any solution to the optimized instance is also a solution to the original instance, and Theorem 26 from Section 7.5 applies. What we have proved above in Lemmas 30 and 31 is that the two optimizations also do not affect the *completeness* of our inference algorithm. Consider the solution $\mathbf{X}$ found by our inference algorithm in Theorem 27. By Lemma 31, solution $\mathbf{X}$ must also satisfy the optimized constraint (7.1'), because $\mathit{DSlice}(\tau, \mathit{Focus}) \subseteq \mathit{Relevant}(\tau, \mathcal{S}*, \mathit{Focus})$ and, by construction, $\mathbf{X}_e = 0$ for all relevant e. And solution $\mathbf{X}$ must also satisfy the optimized constraint (7.5'), because whenever $e \longrightarrow_\tau^* e'$ and $\mathbf{X}_{e'} = 0$, then e' is relevant by construction, e is relevant by Lemma 30, and thus $\mathbf{X}_e = 0$ by construction.

7.6 Experimental Evaluation

In this section, we describe our experimental evaluation of our approach to inferring likely nondeterministic sequential (NDSeq) specifications for parallel programs. In particular, we aim to evaluate the following claim: By examining a small number of representative executions, our specification inference algorithm can automatically generate the correct set of **`if(true*)`** annotations for real Java programs.

To evaluate this claim, we implemented our technique in a prototype tool for Java, called nDetermin, and applied nDetermin to the set of Java benchmarks for which we manually wrote NDSeq specifications in Chapter 6. We compared the quality and accuracy of our automatically-inferred **`if(true*)`**s to the ones in their manually-written NDSeq specifications.

Implementation

Our prototype tool nDetermin uses bytecode instrumentation via `Soot` [150]. nDetermin works in the following four phases:

1. **Annotating focus variables.** In the first phase, the programmer marks the focus lvalues (e.g. objects fields, array elements, local variables, class fields etc.) whose final state compos the program output. We use the simple Java library for annotating focus variables from Chapter 6.

2. **Instrumenting the program.** nDetermin uses `Soot` to inject calls throughout the target program to our analysis code, in order to generate a trace of the events and of the $\longrightarrow$ dependence relation at runtime.

   During the instrumentation phase, we compute control dependencies between statements and identify candidate static blocks that could be annotated with **`if(true*)`**. For this computation, we use off-the-shelf static intra-procedural control flow graph analyses available in `Soot`, as well as the `Indus` program slicer framework [124]. While

Section 7.3 describes our algorithm over a language without unstructured control flow, our implementation handles **`break`**, **`continue`**, early **`return`**, etc.

Note that any Java bytecode instruction that can throw an exception — e.g., a field dereference, an array look-up, a cast, or a division — must be treated as an implicit branch instruction. That is, changing the values flowing into such an instruction can change the control flow by causing or preventing an exception from being thrown. While analyzing the program during the instrumentation phase, our tool identifies control dependencies due to potential branches upon such exceptions.

3. **Collecting runtime information.** We run the program under analysis multiple times (e.g., five in our experiments) and generate parallel execution traces using nDeterMin's instrumentation described above. We use active random testing [132] to generate parallel execution traces.

4. **Inferring the `if(true*)` annotations.** Using the collected runtime information, nDeterMin generates a MinCostSAT instance as described in Section 7.3 and solves the instance using MinCostChaff [68] and MiniSat+ [47]. (nDeterMin can also encode the constraints as a Linear Programming (LP) instance and solve the instance using `lp_solve`[3].) nDeterMin maps the solution of the problem back to the program, pointing to the statements that need to be surrounded by **`if(true*)`**.

Limitations. In Java, it is necessary to handle language features such as objects, exceptions, casts, etc. While our implementation supports many intricacies of the Java language, it has a couple of limitations. First, our implementation tracks neither the shared reads and writes made by uninstrumented native code, nor the flow of data dependence through such native code. Second, in order to reduce the runtime overhead, our tool does not instrument all of the Java standard libraries. Thus, we could miss conflicts or data dependencies carried out through the native code and the Java libraries, and fail to include some events in our inference algorithm. To address the second limitation, for certain shared data structure objects we introduced shared variables and inserted reads or writes of those variables whenever their corresponding objects were accessed. This allowed us to conservatively approximate the conflicts and data dependencies for certain critical standard Java data structures. We did not observe any inaccuracy in our experimental results due to these limitations.

Experimental Setup

We evaluate nDeterMin on the benchmarks previously examined in Chapter 6, which are originally from benchmarks are from the Java Grande Forum (JGF) benchmark suite [46], the Parallel Java (PJ) Library [86], the Lonestar benchmark suite [88], and implementations of

[3] `http://lpsolve.sourceforge.net/`

[145] and [102]. The names and sizes of the benchmarks are listed in Table 7.1. Descriptions of the benchmarks can be found in Section 6.5 in the previous chapter.

Note that we focus here on *parallel* applications, which use multithreading for performance but fundamentally are performing a single computation that can be understood sequentially. We do not consider *concurrent* benchmarks because it is not clear whether or not such programs can be understood sequentially.

For our benchmarks, we use the same focus variable and parallel region annotations as in Chapter 6. (Although these benchmarks are written in a structured parallel style, they use explicit Java threads as Java does not provide `cobegin` or `coforeach` constructs. Thus, the code was annotated to indicate which regions of code correspond to the bodies of structured `cobegin`'s or `coforeach`'s.)

Note also that benchmarks `raytracer` and `phylogeny` both contain parallelism errors. Thus, we apply nDETERMIN to both the original version of each benchmark, and a version in which the error has been fixed. For `raytracer`, we modify a `synchronized` block to use the correct shared lock to protect the key global variable `checksum1`. For `phylogeny`, we make one method `synchronized` in order to eliminate an atomicity error.

We execute each benchmark five times on a single test input, using a simple implementation of race-directed active random testing [132]. For each benchmark, nDETERMIN analyzes all five executions and either infers a placement of **`if(true*)`** for the benchmark's NDSeq specification or reports that the benchmark satisfies no possible NDSeq specification due to a parallel error.

We performed our experiments on a 64-bit Linux machine with a dual Quad-Core/HT Intel(R) Xeon(R) CPU (2.67GHz) processor, 24MB L3 cache and 48GB of DDR3/1066 RAM. For each experiment, we measured the time for solving both SAT instances (without minimizing the number of **`if(true*)`**s) using ZChaff [104] and MiniSat [137], and MinCostSAT instances using MinCostChaff and MiniSat+ generated during the experiment. We observed that for the benchmarks that do not require nondeterministic branches in their NDSeq specifications, the solving time for SAT and MinCostSAT are very close to each other, as the solver can satisfy all the constraints without needing to optimize the number of **`if(true*)`**s. For most of the benchmarks, the solving time was in terms of milliseconds, and few large benchmarks required several seconds to solve the constraints.

Experimental Results

The results of our experimental evaluation are summarized in Table 7.1. The column labeled "All", under "Size of Trace (Events)", reports the number of total events seen in the last execution (of five) of each benchmark, and the column labeled "Sliced Out" reports the number of events removed by our dynamic slicing. nDETERMIN searches for **`if(true*)`** placements to eliminate cycles of transactional conflicts involving the remaining events.

The second-to-last column of Table 7.1 reports the number of **`if(true*)`** constructs inferred for the NDSeq specification for each benchmark. We manually determined whether each of the inferred **`if(true*)`** annotations was correct — i.e., captures all intended non-

Benchmark		Approximate Lines of Code (App + Lib)	# of Parallel Constructs	Size of Manual NDSeq Spec		Size of Trace (Events)		Inferred NDSeq Specification	
				# of `if(*)`	# focus stmts	All	Sliced Out	# of `if(*)`	Correct?
JGF	`sor`	300	1	0	1	905k	561k	0	yes
	`matmult`	700	1	0	1	962k	8k	0	yes
	`series`	800	1	0	5	2008k	1215	0	yes
	`crypt`	1100	2	0	3	493k	100k	0	yes
	`moldyn`	1300	4	0	1	4517k	4300k	0	yes
	`lufact`	1500	1	0	1	1048k	792k	0	yes
	`raytracer`	1900	1	0	1	9125k	8960k	-	-
	`raytracer` (fixed)	1900	1	0	1	9125k	8960k	0	yes
	`montecarlo`	3600	1	0	1	1723k	731k	0	yes
PJ	`pi3`	150 + 15,000	1	0	1	1062k	141	0	yes
	`keysearch3`	200 + 15,000	2	0	4	1062k	1049k	0	yes
	`mandelbrot`	250 + 15,000	1	0	6	576k	330k	0	yes
	`phylogeny`	4400 + 15,000	2	3	8	29k	24k	-	-
	`phylogeny` (fixed)	4400 + 15,000	2	3	8	29k	24k	1	yes
	`stack`	40	1	2	8	1050	356	2	yes
	`queue`	60	1	2	8	325	114	2	yes
	`meshrefine`	1000	1	2	50	930k	845k	2	yes

Table 7.1: Summary of experimental evaluation of our NDSeq specification inference algorithm. All inferred **`if(true*)`** annotations were verified manually to be correct.

determinism, so that the parallel program is equivalent to its NDSeq specification, with no extraneous nondeterminism that would allow the NDSeq version of the program to produce functionally incorrect results. All of the inferred specifications were correct.

For many of the benchmarks, nDetermin correctly infers that no **`if(true*)`** constructs are necessary. All but one of these benchmarks are simply conflict-serializable. As discussed in in Chapter 6, `montecarlo` is not conflict-serializable, but the non-serializable conflicts afftect neither the control flow nor the final result of the program.

For benchmarks `stack`, `queue`, and `meshrefine`, nDetermin infers an NDSeq specification exactly equivalent to the manual specifications from Chapter 6. That is, nDetermin infers the same number of **`if(true*)`** constructs and places them in the same locations as in previous manually-written NDSeq specifications. We note that nDetermin finds specifications slightly smaller than the manual ones, which include a small number of adjacent statements in the **`if(true*)`** that do not strictly need to be enclosed, although in each case the overall behavior of the NDSeq specification is the same whether or not these statements are included in the **`if(true*)`**.

Further, for benchmark `phylogeny` (fixed), while the previous manual NDSeq specification included three **`if(true*)`** constructs, nDetermin correctly infers that only one of these three is actually necessary. The extra **`if(true*)`** appear to have been manually added to

address some possible parallel conflicts that, in fact, can never be involved in non-serializable conflict *cycles*. These two extraneous **if(*true*∗)** do allow the NDSeq specification to perform several nondeterministic behaviors seen during parallel execution of the benchmark. But nDETERMIN correctly determines that these behaviors are possible in the NDSeq specification even without these **if(*true*∗)**.

Note that for two benchmarks, **raytracer** and **phylogeny**, nDETERMIN correctly reports that no NDSeq specification exists — i.e., the SAT instance has no solution (indicated by "-" in Table 7.1). That is, nDETERMIN detects that the events of the dynamic slice (i.e., those not removed by dynamic slicing) are not conflict-serializable. These conflicts exist because both benchmarks contain parallelism errors (atomicity errors due to insufficient synchronization). As a result of these errors, these two parallel applications can produce incorrect results that no sequential version could produce.

Discussion. These experimental results provide promising preliminary evidence for our claim that nDETERMIN can automatically check serializability by way of inferring **if(*true*∗)** necessary for the NDSeq specification of parallel correctness for real parallel Java programs. We believe adding nondeterministic **if(*true*∗)** constructs is the most difficult piece of writing a NDSeq specification, and thus our inference technique can make using NDSeq specifications much easier. Further, such specification inference may allow for fully-automated testing and verification to use NDSeq specifications to separately address parallel and functional correctness.

7.7 Related Work

Several parallel correctness criteria, including data-race freedom [108], atomicity [64], linearizability [100], and determinism [29, 21] have been studied for shared memory parallel programs that separate the reasoning about functionality and parallelism at different granularities of execution. All these criteria provide the separation between parallel and functional correctness partially, as the restriction on thread interleavings is limited, for example, to atomic block boundaries. NDSeq specifications develop this idea up to a complete separation between parallelism and functionality so that the programmer can reason about the intended functionality by examining a sequential, but nondeterministic, program.

Reasoning about conflicting accesses that are simultaneously enabled but ineffective on the rest of the execution is the main challenge in both static [64, 149, 39, 148] and dynamic [164, 160, 97, 12, 62, 74, 153, 91, 26] techniques for checking atomicity and linearizability. Both QED [48] and work on purity [60] for atomicity provide static analyses to rule out spurious warnings due to such conflicts by abstracting these operations to no-ops. Their abstraction techniques resemble identifying irrelevant events by a dependency analysis. However, lack of dynamic information during the static verification is a bottleneck in automating their overall approach

There is a rich literature on generating invariants, temporal specifications, and other kinds of specifications for sequential programs – see Chapter 4 for a discussion of this work.

ALTER [147] is a system in which programmers annotate loops which may be executed nondeterministically and shared reads which may nondeterministically read stale values, in order to enable automatic parallelization of their sequential programs. ALTER [147] also has a test-driven algorithm for *inferring* likely annotations for sequential loops.

Chapter 8

Conclusion

The semiconductor industry has hit the power wall — performance of general-purpose single-core microprocessors can no longer be increased due to power constraints. Instead, the new "Moore's Law" is that the number of cores is doubling every generation, but individual cores are going no faster [10]. Thus, to take advantage of current and future processors, programmers increasingly must write parallel software. But writing correct parallel software remains a challenging task.

Though there have been many advances in techniques to test, debug, and verify parallel software, we have argued that such approaches are often hindered by a lack of any specification from the programmer of the intended, correct parallel behavior of his or her software. And writing formal specifications for software has often been viewed as excessively difficult and time-consuming. Indeed, we have discussed a number of programs for which writing a traditional functional correctness specification was beyond our grasp.

In this dissertation, we have developed novel lightweight specifications that focus on the correctness just of a program's use of parallelism. We have shown that our lightweight specifications enable us to specify, test, debug, and verify the parallelism correctness of a program, without having to grapple with a program's complex and fundamentally-sequential functional correctness.

We have proposed *semantic determinism specifications* for specifying that any run of a parallel program must give semantically-equivalent results, despite the nondeterministic scheduling of the program's parallel threads. We proposed *bridge predicates* — predicates relating pairs of program states from two different program executions — to simply specify, at a high level, what it means for the results of two different parallel executions to be the same. We showed that it was easy to write determinism specifications for a variety of Java benchmarks. We further proposed a simple technique for testing that a parallel program conforms to its determinism specification, by recording and comparing, across pairs of test executions, the program states at the start and end of deterministic blocks. This technique, combined with active testing [132, 85], was effective in finding real parallel determinism errors and separating these errors from many benign program races. Finally, we proposed an algorithm for automatically inferring a likely determinism specification for a program,

given a handful of representative program executions, and showed that our algorithm could automatically generate equivalent or better determinism specifications than those we wrote by hand for nearly all of our benchmarks.

We have proposed *semantic atomicity specifications*, showing that our bridge predicates can be used to generalize traditional atomicity specifications, in order to capture critical high-level noninterference properties of parallel and concurrent code. We showed that it was simple to write semantic atomicity specifications for a number of parallel and concurrent Java benchmarks. We also proposed a technique for testing that a program conforms to its semantic atomicity specification, by generating test parallel test executions in which only a small number of specified atomic blocks overlap, and by testing the semantic strict serializability (i.e., semantic linearizability) of such executions. With our testing technique, we found several known and previously unknown atomicity errors with no false positives, where previous, strict atomicity techniques would have reported many false warnings.

Finally, we have proposed *nondeterministic sequential (NDSeq) specifications* for parallelism correctness, showing that we can give a complete specification for the parallelism correctness of a parallel program with a version of the program that is sequential but contains some limited, controlled nondeterminism. We argued that, if we can verify or otherwise gain confidence in the parallelism correctness of such a program, we can then test, debug, and verify the program's functional correctness on a version of the program with some nondeterminism but with no interleaving of parallel threads. We described several patterns for writing NDSeq specifications and showed that, using our patterns, it was fairly simple to write NDSeq specifications for a number of parallel Java benchmarks. We also proposed a testing technique for checking that an execution of a parallel program conforms to the program's NDSeq specification. Our technique generalizes traditional conflict-serializability checking, by combining the nondeterminism in the specification with a dynamic dependence analysis to safely ignore some conflicting operations. We applied our technique to test our NDSeq specifications, finding the real parallelism errors and eliminating nearly all of the false positives of traditional atomicity checking. We further proposed a technique for automatically inferring a likely NDSeq specification for a parallel program, given a handful of representative executions, and we showed that our hand-written NDSeq specifications could all be reproduced by our inference algorithm. We also proposed two future directions of work for NDSeq specifications: (1) to use our runtime checking technique to classify an erroneous program trace as containing either a sequential or a parallel error, and, in the former case, allowing the error to be debugged on a nondeterministic sequential trace, instead, and (2) to statically verify parallelism correctness, using reduction [93] to show that every parallel program behavior is also a behavior of the NDSeq specification.

Bibliography

[1] Mithun Acharya et al. "Mining API patterns as partial orders from source code: from usage scenarios to specifications". In: *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. ACM, 2007.

[2] Sarita V. Adve et al. "Detecting data races on weak memory systems". In: *18th annual International Symposium on Computer architecture (ISCA)*. Toronto, Ontario, Canada: ACM, 1991, pp. 234–243.

[3] Rahul Agarwal et al. "Optimized run-time race detection and atomicity checking using partial discovered types". In: *20th IEEE/ACM International Conference on Automated software engineering (ASE)*. Long Beach, CA: ACM, 2005, pp. 233–242.

[4] Gul Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[5] Hiralal Agrawal and Joseph R. Horgan. "Dynamic program slicing". In: *Programming Language Design and Implementation (PLDI)*. 1990.

[6] R. Alur et al. "Synthesis of Interface Specifications for Java Classes". In: *Proceedings of POPL'05 (32nd ACM Symposium on Principles of Programming Languages)*. 2005.

[7] Glenn Ammons, Rastislav Bodik, and James R. Larus. "Mining specifications". In: *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Jan. 2002.

[8] Z. Anderson et al. "SharC: checking data sharing strategies for multithreaded C". In: *ACM SIGPLAN conference on Programming language design and implementation (PLDI 2008)*. ACM. 2008, pp. 149–158.

[9] C. Artho, K. Havelund, and A. Biere. "High-level data races". In: *Software Testing Ver. and Reliability* 13.4 (2003), pp. 207–227.

[10] Krste Asanovic et al. *The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View*. Tech. rep. UCB/EECS-2008-23. EECS Department, University of California, Berkeley, 2008.

[11] A. Aviram et al. "Efficient system-enforced deterministic parallelism". In: *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (ODSI)*. USENIX Association. 2010, pp. 1–16.

[12] P. Madhusudan Azadeh Farzan. "Monitoring Atomicity in Concurrent Programs". In: *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*. Springer, 2008, pp. 52–65.

[13] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2001, pp. 268–283.

[14] G. Barnes. "A method for implementing lock-free shared-data structures". In: *5th ACM symposium on Parallel Algorithms and Architectures (SPAA)*. 1993.

[15] Tom Bergan et al. "CoreDet: a compiler and runtime system for deterministic multithreaded execution". In: *Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '10. ACM, 2010, pp. 53–64.

[16] Emery D. Berger et al. "Grace: safe multithreaded programming for C/C++". In: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. OOPSLA '09. ACM, 2009, pp. 81–96.

[17] Dirk Beyer et al. "The software model checker Blast: Applications to software engineering". In: *International Journal on Software Tools for Technology Transfer* 9 (5 2007), pp. 505–525.

[18] Stephen M. Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2006, pp. 169–190.

[19] R.D. Blumofe et al. "Cilk: An efficient multithreaded runtime system". In: *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP)*. 1995.

[20] R. Bocchino et al. "Parallel Programming Must Be Deterministic by Default". In: *First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009)*. 2009.

[21] Robert L. Bocchino Jr. et al. "A type and effect system for Deterministic Parallel Java". In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2009, pp. 97–116.

[22] Robert L. Bocchino Jr. et al. "Safe nondeterminism in a deterministic-by-default parallel language". In: *Principles of Programming Languages (POPL)*. 2011, pp. 535–548.

[23] Chandrasekhar Boyapati and Martin C. Rinard. "A Parameterized Type System for Race-Free Java Programs". In: *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*. 2001, pp. 56–69.

[24] D. Bruening. "Systematic testing of multithreaded Java programs". MA thesis. MIT, 1999.

[25] Sebastian Burckhardt et al. "A randomized scheduler with probabilistic guarantees of finding bugs". In: *ASPLOS '10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*. Pittsburgh, Pennsylvania, USA: ACM, 2010.

[26] Sebastian Burckhardt et al. "Line-Up: A complete and automatic linearizability checker". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010, pp. 330–340.

[27] Jacob Burnim, George Necula, and Koushik Sen. "Separating Functional and Parallel Correctness using Nondeterministic Sequential Specifications". In: *HotPar '10: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA, USA: USENIX, 2010.

[28] Jacob Burnim, George Necula, and Koushik Sen. "Specifying and Checking Semantic Atomicity for Multithreaded Programs". In: *ASPLOS '11: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, California, USA: ACM, 2011, pp. 79–90.

[29] Jacob Burnim and Koushik Sen. "Asserting and checking determinism for multithreaded programs". In: *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. Amsterdam, The Netherlands: ACM, 2009, pp. 3–12.

[30] Jacob Burnim and Koushik Sen. "Asserting and Checking Determinism for Multithreaded Programs". In: *Communications of the ACM (CACM)* 53 (6 June 2010), pp. 97–105.

[31] Jacob Burnim and Koushik Sen. "DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs". In: *ICSE '10: Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering*. Cape Town, South Africa: ACM, 2010, pp. 415–424.

[32] Jacob Burnim, Koushik Sen, and Christos Stergiou. "Testing Concurrent Programs on Relaxed Memory Models". In: *ISSTA '11: Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Toronto, Ontario, Canada: ACM, 2011, pp. 122–132.

[33] Jacob Burnim et al. "NDetermin: Inferring Nondeterministic Sequential Specifications for Parallelism Correctness (poster)". In: *PPoPP '12: Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*. (to appear). New Orleans, LA, USA: ACM, 2012.

[34] Jacob Burnim et al. "NDSeq: Runtime Checking for Nondeterministic Sequential Specifications of Parallel Correctness". In: *PLDI '11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Jose, California, USA: ACM, 2011, pp. 401–414.

[35] Richard H. Carver and Yu Lei. "A General Model for Reachability Testing of Concurrent Programs". In: *6th International Conference on Formal Engineering Methods (ICFEM'04)*. Vol. 3308. LNCS. 2004, pp. 76–98.

[36] S. Chaki et al. "Modular verification of software components in C". In: *IEEE Transactions on Software Engineering (TSE)* 30.6 (2004), pp. 388–402.

[37] Lee Chew and David Lie. "Kivati: fast detection and prevention of atomicity violations". In: *Proceedings of the 5th ACM European conference on computer systems*. EuroSys '10. 2010, pp. 307–320.

[38] J. D. Choi et al. "Efficient and precise datarace detection for multithreaded object-oriented programs". In: *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*. 2002, pp. 258–269.

[39] Robert Colvin et al. "Formal verification of a lazy concurrent list-based set algorithm". In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*. Springer, 2006, pp. 475–488.

[40] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. "DySy: Dynamic symbolic execution for invariant inference". In: *30th ACM/IEEE International Conference on Software Engineering (ICSE)*. 2008.

[41] L. Dagum, R. Menon, and S.G. Inc. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science & Engineering* 5.1 (1998), pp. 46–55.

[42] Joseph Devietti et al. "DMP: deterministic shared memory multiprocessing". In: *ACM conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009.

[43] Joseph Devietti et al. "RCDC: a relaxed consistency deterministic computer". In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ASPLOS '11. ACM, 2011, pp. 67–78.

[44] A. Dinning and E. Schonberg. "Detecting access anomalies in programs with critical sections". In: *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*. 1991.

[45] O. Edelstein et al. "Multithreaded Java program test generation". In: *IBM Systems Journal* 41.1 (2002), pp. 111–125.

[46] Edinburgh Parallel Computing Centre. *Java Grande Forum Benchmark Suite*. `www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`.

[47] Niklas Eén and Niklas Sörensson. "Translating Pseudo-Boolean Constraints into SAT". In: *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 2.3-4 (2006), pp. 1–25.

[48] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "A calculus of atomic actions". In: *Principles of Programming Languages (POPL)*. 2009, pp. 2–15.

[49] Dawson R. Engler and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *19th ACM Symposium on Operating Systems Principles (SOSP)*. 2003.

[50] Michael D. Ernst et al. "Quickly Detecting Relevant Program Invariants". In: *Proceedings of the 22nd International Conference on Software Engineering*. 2000, pp. 449–458.

[51] Eitan Farchi, Yarden Nir, and Shmuel Ur. "Concurrent Bug Patterns and How to Test Them". In: *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003.

[52] A. Farzan, P. Madhusudan, and F. Sorrentino. "Meta-analysis for atomicity violations under nested locking". In: *Computer Aided Verification*. Springer. 2009, pp. 248–262.

[53] A. Finkel, B. Willems, and P. Wolper. "A direct symbolic approach to model checking pushdown systems". In: *Workshop on Verification of Infinite State Systems (INFINITY)*. 1997.

[54] C. Flanagan and S. N. Freund. "Detecting race conditions in large programs". In: *Proc. of the Program Analysis for Software Tools and Engineering Conference*. 2001.

[55] C. Flanagan et al. "Extended static checking for Java". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2002.

[56] Cormac Flanagan. "Verifying Commit-Atomicity Using Model-Checking". In: *11th International SPIN Workshop*. 2004, pp. 252–266.

[57] Cormac Flanagan and Stephen N. Freund. "Atomizer: a dynamic atomicity checker for multithreaded programs". In: *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2004.

[58] Cormac Flanagan and Stephen N. Freund. "FastTrack: efficient and precise dynamic race detection". In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '09. ACM, 2009, pp. 121–133.

[59] Cormac Flanagan and Stephen N. Freund. "Type-based race detection for Java". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 2000.

[60] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. "Exploiting purity for atomicity". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2004, pp. 221–231.

[61] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. "Exploiting Purity for Atomicity". In: *IEEE Transactions on Software Engineering* 31.4 (Apr. 2005), pp. 275–291.

[62] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs". In: *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2008, pp. 293–303.

[63] Cormac Flanagan and Rustan M. Leino. "Houdini, an Annotation Assistant for ESC/Java". In: *Proceedings of the International Symposium of Formal Methods Europe (FME)*. 2001.

[64] Cormac Flanagan and Shaz Qadeer. "A type and effect system for atomicity". In: *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 2003.

[65] Cormac Flanagan and Shaz Qadeer. "Types for atomicity". In: *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*. New Orleans, Louisiana, USA: ACM, 2003, pp. 1–12.

[66] R.W. Floyd. "Assigning meanings to programs". In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[67] P. Fonseca, C. Li, and R. Rodrigues. "Finding complex concurrency bugs in large multi-threaded applications". In: *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS)*. 2011, pp. 215–228.

[68] Zhaohui Fu and Sharad Malik. "Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2006.

[69] M. Gabel and Z. Su. "Javert: fully automatic mining of general temporal properties from dynamic traces". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM. 2008, pp. 339–349.

[70] M. Gabel and Z. Su. "Online inference and enforcement of temporal properties". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM. 2010, pp. 15–24.

[71] David Gay et al. "The nesC language: A holistic approach to networked embedded systems". In: *ACM SIGPLAN Conference on Programming language design and implementation*. 2003, pp. 1–11.

[72] Patrice Godefroid. "Model Checking for Programming Languages using Verisoft". In: *24th Symposium on Principles of Programming Languages*. 1997, pp. 174–186.

[73] Claire Goues and Westley Weimer. "Specification Mining with Few False Positives". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2009.

[74] Christian Hammer et al. "Dynamic detection of atomic-set-serializability violations". In: *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 231–240.

[75] Sudheendra Hangal and Monica S. Lam. "Tracking Down Software Bugs Using Automatic Anomaly Detection". In: *Proceedings of the International Conference on Software Engineering*. 2002.

[76] J. Hatcliff, Robby, and M. B. Dwyer. "Verifying atomicity specifications for concurrent object-oriented software using model-checking". In: *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*. 2004, pp. 175–190.

[77] Maurice Herlihy and Eric Koskinen. "Transactional boosting: a methodology for highly-concurrent transactional objects". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, USA: ACM, 2008, pp. 207–216.

[78] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[79] D.R. Hower et al. "Calvin: Deterministic or not? free will to choose". In: *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2011, pp. 333–334.

[80] IEEE. "POSIX Part 1: System API- Amend. 1: Realtime Extension [C Language]". In: *IEEE Std 1003.1b-1993* (1994).

[81] Intel®. *Threading Building Blocks for Open Source*. `http://threadingbuildingblocks.org/`.

[82] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. "OoOJava: software out-of-order execution". In: *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*. PPoPP '11. ACM, 2011, pp. 57–68.

[83] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in dataflow programming languages". In: *ACM Comput. Surv.* 36.1 (2004), pp. 1–34.

[84] Pallavi Joshi et al. "A randomized dynamic program analysis technique for detecting real deadlocks". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 2009.

[85] Pallavi Joshi et al. "An extensible active testing framework for concurrent programs". In: *21st International Conference on Computer Aided Verification (CAV'09)*. Lecture Notes in Computer Science. Springer, 2009.

[86] Alan Kaminsky. "Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java". In: *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2007.

[87] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. "Coarse-grained transactions". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2010, pp. 19–30.

[88] M. Kulkarni et al. "Lonestar: A suite of parallel irregular programs". In: *International Symposium on Performance Analysis of Systems and Software, (ISPASS)*. 2009.

[89] Milind Kulkarni et al. "Exploiting the Commutativity Lattice". In: *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '11. 2011.

[90] Milind Kulkarni et al. "Optimistic parallelism requires abstractions". In: *ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2007, pp. 211–222.

[91] Zhifeng Lai, S. C. Cheung, and W. K. Chan. "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 235–244.

[92] E. A. Lee. "The problem with threads". In: *Computer* 39.5 (2006), pp. 33–42.

[93] R. J. Lipton. "Reduction: A method of proving properties of parallel programs". In: *Communications of the ACM* 18.12 (1975), pp. 717–721.

[94] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "Dthreads: efficient deterministic multithreading". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. ACM, 2011, pp. 327–336.

[95] Francesco Logozzo. "Automatic Inference of Class Invariants". In: *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*. 2004.

[96] H.W. Loidl et al. "Comparing parallel functional languages: Programming and performance". In: *Higher-Order and Symbolic Computation* 16.3 (2003), pp. 203–251.

[97] Shan Lu et al. "AVIO: Detecting atomicity violations via access interleaving invariants". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2006.

[98] Muhammad Zubair Malik et al. "Generating Representation Invariants of Structurally Complex Data". In: *TACAS*. 2007, pp. 34–49.

[99] Nir Shavit Maurice Herlihy. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, Inc., 2008.

[100] Jeannette M. Wing Maurice P. Herlihy. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (3 1990), pp. 463–492.

[101] B. Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[102] Maged M. Michael and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing (PDOC)*. Philadelphia, Pennsylvania, United States: ACM, 1996, pp. 267–275.

[103] N. Mittal and V. K. Garg. "Consistency Conditions for Multi-Object Distributed Operations". In: *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 1998, pp. 582–.

[104] M.W. Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference (DAC)*. ACM. 2001, pp. 530–535.

[105] M. Musuvathi and S. Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: *ACM Symposium on Programming Language Design and Implementation (PLDI'07)*. 2007.

[106] Madanlal Musuvathi et al. "Finding and Reproducing Heisenbugs in Concurrent Programs". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. 2008.

[107] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2006.

[108] Robert H. B. Netzer and Barton P. Miller. "What are race conditions?: Some issues and formalizations". In: *ACM Lett. Prog. Lang. Syst.* 1.1 (1992), pp. 74–88.

[109] Yang Ni et al. "Open nesting in software transactional memory". In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2007, pp. 68–78.

[110] A. Nistor, D. Marinov, and J. Torrellas. "InstantCheck: Checking the determinism of parallel programs using on-the-fly incremental hashing". In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society. 2010, pp. 251–262.

[111] Robert O'Callahan and Jong-Deok Choi. "Hybrid dynamic data race detection". In: *ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2003.

[112] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. "Kendo: Efficient Deterministic Multithreading in Software". In: *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009.

[113] Susan Owicki and David Gries. "Verifying properties of parallel programs: an axiomatic approach". In: *Commun. ACM* 19.5 (1976), pp. 279–285.

[114] Christos Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., 1986.

[115] Christos H. Papadimitriou. "The serializability of concurrent database updates". In: *Journal of the ACM (JACM)* 26.4 (Oct. 1979), pp. 631–653.

[116] Chang-Seo Park and Koushik Sen. "Randomized active atomicity violation detection in concurrent programs". In: *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Atlanta, Georgia, 2008.

[117] Soyeon Park, Shan Lu, and Yuanyuan Zhou. "CTrigger: exposing atomicity violation bugs from their hiding places". In: *ASPLOS '09: Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems*. Washington, DC, USA: ACM, 2009, pp. 25–36.

[118] Keshav Pingali et al. "The tao of parallelism in algorithms". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '11. ACM, 2011, pp. 12–25.

[119] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. "Safe programmable speculative parallelism". In: *Programming Language Design and Implementation (PLDI)*. 2010, pp. 50–61.

[120] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. "LOCKSMITH: context-sensitive correlation analysis for race detection". In: *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2006.

[121] Christoph von Praun, Luis Ceze, and Calin Caşcaval. "Implicit parallelism with ordered transactions". In: *Principles and Practice of Parallel Programming (PPoPP)*. 2007, pp. 79–89.

[122] Christoph von Praun and Thomas R. Gross. "Object race detection". In: *16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2001.

[123] G. Ramalingam. "Context-sensitive synchronization-sensitive analysis is undecidable". In: *ACM Trans. Prog. Lang. Syst.* 22.2 (2000), pp. 416–430.

[124] Venkatesh Prasad Ranganath and John Hatcliff. "Slicing concurrent Java programs using Indus and Kaveri". In: *International Journal on Software Tools for Technology Transfer* 9 (5 2007), pp. 489–504.

[125] Lawrence Rauchwerger and David Padua. "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization". In: *Programming Language Design and Implementation (PLDI)*. 1995, pp. 218–232.

[126] Martin C. Rinard and Pedro C. Diniz. "Commutativity analysis: A new analysis framework for parallelizing compilers". In: *Programming Language Design and Implementation (PLDI)*. 1996, pp. 54–67.

[127] David S. Rosenblum. "Towards a method of programming with assertions". In: *ICSE '92: Proceedings of the 14th international conference on Software engineering*. Melbourne, Australia: ACM, 1992, pp. 92–104.

[128] C. Sadowski, S.N. Freund, and C. Flanagan. "SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs". In: *18th European Symposium on Programming (ESOP)*. 2009.

[129] J.H. Saltz, R. Mirchandaney, and K. Crowley. "Run-time parallelization and scheduling of loops". In: *Computers, IEEE Transactions on* 40.5 (1991), pp. 603 –612.

[130] Stefan Savage et al. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". In: *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997), pp. 391–411.

[131] Koushik Sen. "Effective Random Testing of Concurrent Programs". In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 2007.

[132] Koushik Sen. "Race Directed Random Testing of Concurrent Programs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 2008.

[133] Ohad Shacham et al. "Testing atomicity of composed concurrent operations". In: *Proceedings of the 2011 ACM international conference on Object Oriented Programming Systems, Languages, and Applications*. OOPSLA '11. ACM, 2011, pp. 51–64.

[134] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Principles of Distributed Computing (PODC)*. 1995, pp. 204–213.

[135] Stephen F. Siegel et al. "Combining symbolic execution with model checking to verify parallel numerical programs". In: *ACM Transactions on Software Engineering and Methodology* 17.2 (2008), pp. 1–34.

[136] Yannis Smaragdakis et al. "Sound predictive race detection in polynomial time". In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. POPL '12. ACM, 2012, pp. 387–400.

[137] N. Sorensson and N. Een. "Minisat v1.13 – a SAT solver with conflict-clause minimization". In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Vol. 3569. Lecture Notes in Computer Science. Springer, June 2005.

[138] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. "PENELOPE: weaving threads to expose atomicity violations". In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of Software Engineering*. FSE '10. ACM, 2010, pp. 37–46.

[139] N. Sterling. "Warlock: A static data race analysis tool". In: *USENIX Winter Technical Conference*. 1993, pp. 97–106.

[140] Scott D. Stoller. "Testing Concurrent Java Programs using Randomized Scheduling". In: *2nd Workshop on Runtime Verification (RV)*. 2002.

[141] W. Sumner, C. Hammer, and J. Dolby. "Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs". In: *Proceedings of the 2nd International Conference on Runtime Verification (RV)*. Springer. 2011, pp. 161–176.

[142] Mana Taghdiri and Daniel Jackson. "Inferring specifications to detect errors in code". In: *Automated Software Engineering* 14.1 (2007), pp. 87–121.

[143] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A language for streaming applications". In: *11th International Conference on Compiler Construction (CC)*. 2002.

[144] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. "Discovering Likely Method Specifications". In: *ICFEM*. 2006, pp. 717–736.

[145] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Tech. rep. RJ 5118. IBM Almaden Research Center, Apr. 1986.

[146] Omer Tripp et al. "HAWKEYE: effective discovery of dataflow impediments to parallelization". In: *Proceedings of the 2011 ACM international conference on Object Oriented Programming Systems, Languages, and Applications*. OOPSLA '11. 2011.

[147] Abhishek Udupa, Kaushik Rajan, and William Thies. "ALTER: exploiting breakable dependences for parallelization". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '11. 2011.

[148] Viktor Vafeiadis. "Shape-Value Abstraction for Verifying Linearizability". In: *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2009, pp. 335–348.

[149] Viktor Vafeiadis et al. "Proving correctness of highly-concurrent linearisable objects". In: *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2006, pp. 129–136.

[150] Raja Vallee-Rai et al. "Soot - a Java Optimization Framework". In: *CASCON 1999*. 1999, pp. 125–135.

[151] Mandana Vaziri, Frank Tip, and Julian Dolby. "Associating synchronization constraints with data in an object-oriented language". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2006, pp. 334–345.

[152] Pavol Černý et al. "Model Checking of Linearizability of Concurrent List Implementations". In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. Springer, 2010, pp. 465–479.

[153] Martin Vechev, Eran Yahav, and Greta Yorsh. "Experience with Model Checking Linearizability". In: *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Springer, 2009, pp. 261–278.

[154] Martin Vechev et al. "Verifying Determinism of Structured Parallel Programs". In: *Static Analysis Symposium (SAS)*. 2010.

[155] W. Visser et al. "Model Checking Programs". In: *15th International Conference on Automated Software Engineering (ASE)*. IEEE, 2000.

[156] W. Visser et al. "Model checking programs". In: *Automated Software Engineering* 10.2 (2003), pp. 203–232.

[157] C. Wang et al. "Trace-based symbolic analysis for atomicity violations". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2010, pp. 328–342.

[158] L. Wang and S. D. Stoller. "Run-Time Analysis for Atomicity." In: *3rd Workshop on Run-time Verification (RV'03)*. Vol. 89. ENTCS 2. 2003.

[159] Liqiang Wang and Scott D. Stoller. "Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs". In: *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, Mar. 2006, pp. 137–146.

[160] Liqiang Wang and Scott D. Stoller. "Runtime Analysis of Atomicity for Multithreaded Programs". In: *IEEE Transactions on Software Engineering* 32 (2 2006), pp. 93–110.

[161] Andrzej Wasylkowski and Andreas Zeller. "Mining Temporal Specifications from Object Usage". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. 2009, pp. 295–306.

[162] Y. Wei et al. "Inferring better contracts". In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM. 2011, pp. 191–200.

[163] J. Whaley, M. C. Martin, and M. S. Lam. "Automatic Extraction of Object-Oriented Component Interfaces". In: *Proceedings of ACM SIGSOFT ISSTA'02 (International Symposium on Software Testing and Analysis)*. 2002.

[164] Jeannette M. Wing and Chun Gong. "Testing and verifying concurrent objects". In: *Journal of Parallel and Distributed Computing* 17.1-2 (Jan. 1993), pp. 164–182.

[165] Jinlin Yang and David Evans. "Dynamically inferring temporal properties". In: *PASTE '04*. Washington DC, USA: ACM, 2004, pp. 23–28.

[166] Jinlin Yang et al. "Perracotta: mining temporal API rules from imperfect traces". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. ACM, 2006, pp. 282–291.

[167] Wei Zhang, Chong Sun, and Shan Lu. "ConMem: detecting severe concurrency bugs through an effect-oriented approach". In: *ASPLOS '10: Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems*. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 179–192.

[168] Wei Zhang et al. "ConSeq: Detecting Concurrency Bugs through Sequential Errors". In: *ASPLOS '11: Proceeding of the 16th international conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2011.